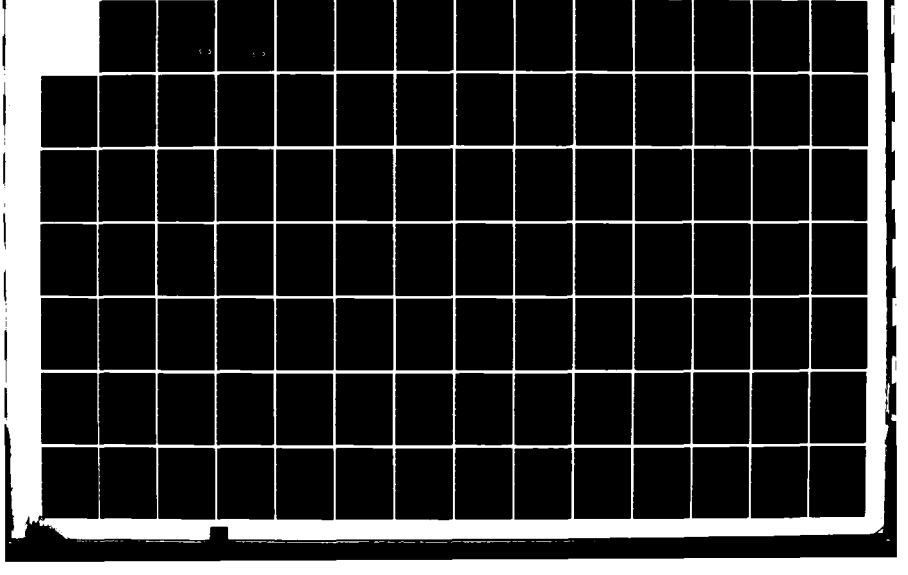
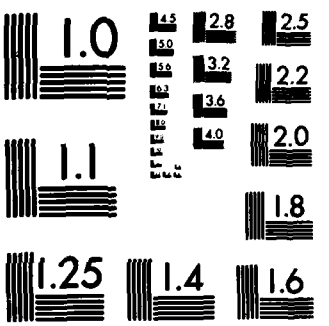
AD-A150 612 THE HIERARCHICAL DATABASE DECOMPOSITION APPROACH TO DATABASE CONCURRENCY..(U) ALFRED P SLOAN SCHOOL OF MANAGEMENT CAMBRIDGE MA CENTER FOR I.. M HSU DEC 84 1/4

UNCLASSIFIED CISR-TR-16 N00039-83-C-0463 F/G 9/2 NL





(12)

The Hierarchical Database Decomposition Approach to Database Concurrency Control

Technical Report #16

Meichun Hsu

December 1984

Center for Information Systems Research
Massachusetts Institute of Technology
Sloan School of Management
77 Massachusetts Avenue
Cambridge, Massachusetts, 02139

Contract Number N00039-83-C-0463
Internal Report Number M010-8312-16
Deliverable Numbers 1 & 2

The Hierarchical Database Decomposition Approach to Database Concurrency Control

Technical Report #16

Meichun Hsu

December 1984

Principal Investigator:
Professor Stuart E. Madnick

Prepared for:
Naval Electronics Systems Command
Washington, D.C.

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER Technical Report #	2. GOVT ACCESSION NO. AD-A150 612	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) The Hierarchical Database Decomposition Approach to Database Concurrency Control		5. TYPE OF REPORT & PERIOD COVERED
		6. PERFORMING ORG. REPORT NUMBER M010-8312-
7. AUTHOR(s) Meichun Hsu		8. CONTRACT OR GRANT NUMBER(s)
9. PERFORMING ORGANIZATION NAME AND ADDRESS Sloan School of Management Massachusetts Institute of Technology Cambridge MA 02139		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS		12. REPORT DATE December 1983
		13. NUMBER OF PAGES 287
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office)		15. SECURITY CLASS. (of this report) Unclassified
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Database management systems, concurrency control, database computer

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

A generally accepted criterion for correctness of database concurrency control is the criterion of serializability of transactions. The classical approaches to enforcing serializability are the two-phase locking and the timestamping methods. Either approach requires that a read operation from a transaction be registered in the form of a read timestamp or a read lock

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE
S/N 0102-014-6601

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

so that a write operation from a concurrent transaction will not interfere improperly with the read operation. However, setting a lock or leaving a timestamp with a data element is an expensive operation. The purpose of the current research is to seek ways to reduce the overhead of synchronization of certain types of read accesses, and at the same time achieving the goal of serializability.

To this end, a hierarchical structure is proposed here as the means for analyzing opportunities of reducing concurrency control overhead in a database application. A theorem is proven which indicates that in a hierarchically decomposed database one can construct yet a third type of partial order of all transactions which is different from the simple commit order and the simple initiation order that hve been used in the existing algorithms. This new type of partial order, called topologically follows, enables a new algorithm for concurrency control to be devised which is belived to have the potential of reducing the overhead of read access synchronization in a database system.

The report develops the theory that supports the correctness of such analysis and algorithms. An implementation scheme for the proposed algorithm, the Hierarchical Timestamping Algorithm, is described. The complexity of the hierarchical database decomposition methodology is analyzed and a heuristics procedure is proposed. The HDD approach is applied to three different application areas in which its effect on relieving database contention and on structuring databases and transactions so as to reduce concurrency control overhead is demonstrated.

THE HIERARCHICAL DATABASE DECOMPOSITION APPROACH TO DATABASE CONCURRENCY CONTROL

by

Meichun Hsu

Submitted to the Alfred P. Sloan School of Management
on October 1983 in partial fulfillment of the
requirements for the degree of Doctor of Philosophy

ABSTRACT

Increasing demand for information system capacity has prompted researchers to find ways to improve the computer systems used in information processing. Database management systems (DBMS's) represent one such effort to provide better information services at lower costs. In order to minimize response time and maximize throughput, it is desirable that a DBMS supports multiple users at the same time, allowing multiple transactions to run in parallel. However, for the purpose of maintaining database consistency and integrity, such parallelism must be properly controlled. For example, suppose two transactions that transfer money into the same bank account are run in parallel. Without proper coordination between the two transactions, it is possible that both of them would read the same old balance, modify it independently and write these independently-modified balances back to the database. If this happens, the final balance will reflect the result of only one, but not both, of the transactions, causing one transaction to be lost. To prevent such violation of database consistency and integrity from taking place, the concurrency control facility is an indispensable component of the database management system.

A generally accepted criterion for correctness of a concurrency control algorithm is the criterion of *serializability* of transactions. The classical approaches to enforcing serializability are the *two-phase locking* technique and the *timestamp ordering* technique. The first technique ensures serializability by imposing a partial order on all transactions based on their commit order, while the second on their initiation order. Either approach requires that a read operation from a transaction be *registered* (in the form of either a read timestamp or a read lock), so that a write operation from a concurrent transaction will not interfere improperly with the read operation. However, setting a lock or leaving a timestamp with a data element is an expensive operation. The purpose of the current research is to seek ways to reduce the overhead of synchronizing certain types of read accesses, and at the same time achieving the goal of serializability.

To this end, a hierarchical structure is proposed here as the means for analyzing opportunities of reducing concurrency overhead in a database application. A theorem is discovered which indicates that in a hierarhically decomposed database one can construct yet a third type of partial order of all

transactions which is different from the simple commit order and the simple initiation order that have been used in the existing algorithms. This new type of partial order, called *topologically follows*, enables a new algorithm for concurrency control to be devised which is believed to have the ability of effectively reducing the overhead of read access synchronization in a database system that endorses a hierarchical decomposition of the database.

The thesis develops the theory that supports the correctness of such analysis and algorithms. An implementation scheme for the proposed algorithm, the Hierarchical Timestamping Algorithm, is described. The complexity of the hierarchical database decomposition methodology is analyzed and a heuristics procedure is proposed. The HDD approch is applied to three different application areas in which its effect on relieving database contention and on structuring databases and transactions so as to reduce concurrency control overhead is demonstrated.

Thesis Supervisor: Prof. Stuart E. Madnick
Associate Professor of Management
M.I.T. Sloan School of Management

ACKNOWLEDGEMENT

I am deeply grateful to my advisor, Professor Stuart Madnick, for his continuous guidance and support. He has been to me the greatest teacher who sets high standards and does not hesitate in providing whatever help and spiritual and intellectual lifts his students need. I consider myself truely fortunate and most privileged to be his student.

My thesis committe members, Professor Little, Dr. Frank and Professor Papadimitriou have all at various stages provided helpful suggestions. I am especially thankful to Dr.Frank who patiently walked through many ill-formed ideas with me and helped kept me on the right track.

All members of the INFOPLEX research group, through regular seminars and informal conversations, have given me ample intellectual stimulation. In particular, I would like to thank Allen Moulton for being both an articulate critic and a faithful friend.

My appreciation also goes to my colleague and friend Dr. Arvola Chan who was instrumental in getting me out of some technical quagmire at an early stage of the game and has stayed supportive ever since.

My personal friend, Nico Spinelli, has provided me with an abundance of moral support and encouragement throughout my study at M.I.T. and especially during the difficult times at the thesis stage.

Finally, my family members have been behind me and shared much of the ups and downs with me. They have made this formidable task so much more meaningful to me.

Research reported in this thesis has been supported, in part, by the Naval Electronics Systems Command through contract N0039-81-c-0663.

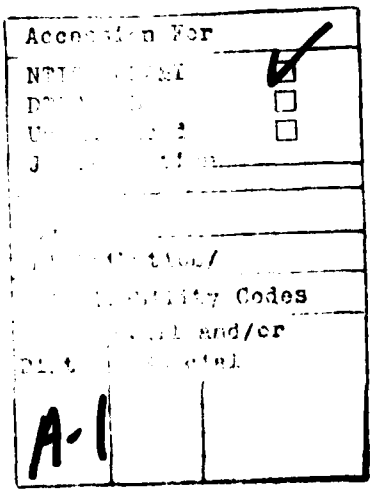


TABLE OF CONTENTS

LIST OF ILLUSTRATIONS

1.0 INTRODUCTION AND THE SCOPE OF THE THESIS

1.1 INTRODUCTION TO THE CONCURRENCY CONTROL PROBLEM

Increasing demand for information system capacity has prompted researchers to find ways to improve the computer systems used in information processing. Database management systems (DBMS's) represent one such effort to provide better information services at lower costs. In order to minimize response time and maximize throughput, it is desirable that a DBMS supports multiple users at the same time, allowing multiple transactions to run in parallel. However, for the purpose of maintaining database consistency and integrity, such parallelism must be properly controlled. Therefore the concurrency control facility is an indispensable component of the database management system.

The role of a concurrency control mechanism is to preserve the atomicity of a user transaction, i.e., it will prevent the processing of a transaction from being *erroneously interleaved* with other concurrenct transactions, so that each transaction sees a consistent database state and, if necessary, can be recovered or backed out as a single unit.

A typical example of database inconsistency induced by improper interleaving of steps of concurrenct transactions is shown in Figure 1.

Currently there is $100 in Smith's Account.

t1: Deposit $50 into Smith's account

t2: Withdraw $50 from Smith's account.

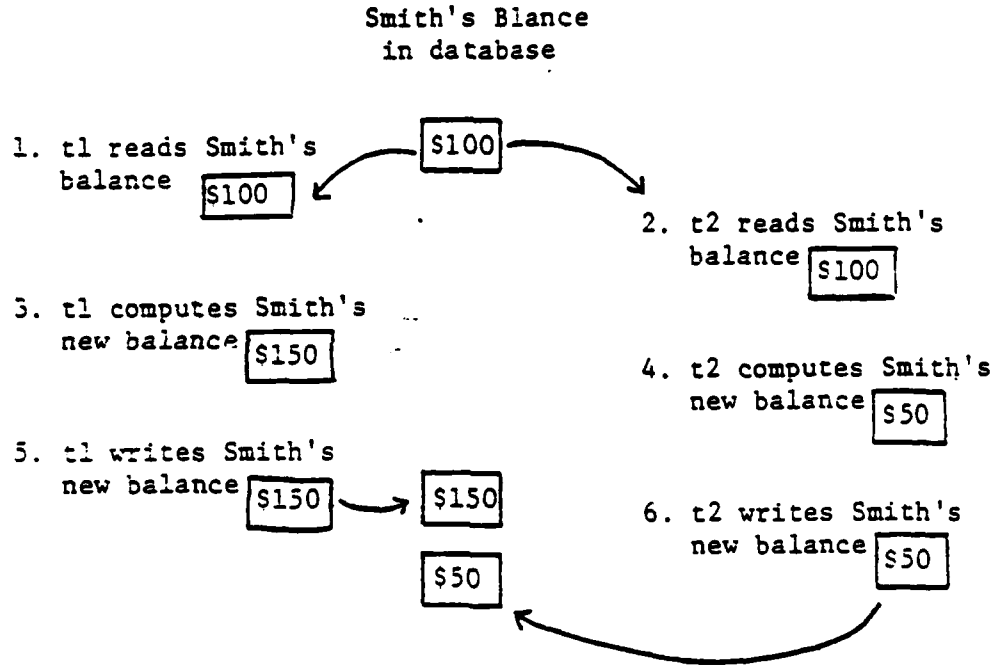


Figure 1. An example of database inconsistency induced by concurrent processing.

As shown in the figure, two transactions simultaneously accessing the same piece of data may result in lost update, leaving the database in an incorrect state. This is due to the fact that the database access steps from these two transactions are not properly interleaved. By requiring that the database management system exercise control over such interleaving of concurrent transactions, the undesirable effects can be eliminated.

1.2 THE SCOPE OF CURRENT RESEARCH

1.2.1 INTRODUCTION

A generally accepted criterion for correctness of a concurrency control algorithm is the criterion of serializability of transactions, which means that interleaving is harmless so long as one can show that the net effect of such interleaving is *equivalent to some serialized processing* (i.e., one after another) of all the transactions involved. In the above example, it is apparent that the steps of the two transactions are scheduled in such a way that there exists no serialized schedule (i.e., either t_1 after t_2 or t_2 after t_1) that would have generated the same net effect. Therefore the schedule of these steps is not serializable, and therefore incorrect.

The classical approach to enforcing serializability is the *two-phase locking* technique. This technique locks up data elements being accessed by one transaction and blocks other transactions from operating on these data elements until the first transaction is finished. Another approach to dealing with this problem is that of the *timestamp ordering technique*, which stamps the data elements with the timestamps of the transactions that have operated on them so as to prevent violation of a pre-determined order from taking place.

Either approach requires that a read operation from a transaction be *registered* (in the form of either a read timestamp or a read lock), so that a write operation from a concurrent transaction will not interfere improperly with the read operation. Setting a lock or leaving a timestamp with a data element is an expensive operation. It not only incurs a write operation in the database (in the form of setting the read lock or writing the timestamp), but also potentially causes delays for concurrent transactions that might be avoided.

The purpose of the current research is to seek ways to reduce the overhead of synchronizing certain types of read accesses, and at the same time achieving the goal of serializability. A more formal and comprehensive overview will be given in the next chapter of the literature in concurrency control, including efforts up to now that have aimed at similar goals. Here we intuitively motivate this research further through a brief discussion of some empirical observations of the impact

of concurrency control on performance of database systems and an example that illustrates the scope of this thesis research.

1.2.2 CONCURRENCY CONTROL AS A POTENTIAL BOTTLENECK OF DBMS'S

It was previously pointed out that setting locks and leaving timestamps are potentially expensive operations. We now present a more specific discussion on how the concurrency control activities could be a threat to performance. We will approach this issue from two different perspectives. The first is the impact of the processing overhead of the concurrency control activities on the effective level of multiprocessing. The second is their impact on the level of transaction concurrency achievable and on overall performance.

1.2.2.1 PROCESSING OVERHEAD OF CONCURRENCY CONTROL

In <Gray78> it was reported that, in System R, the lock manager comprises 3 percent of the code and consumes about 10 percent of the instruction execution time. The latter obviously would vary a great deal depending on applications. A lock-unlock pair typically costs 350 instructions. No comparable statistics on timestamping algorithm have been reported, but it is believed that the overhead of timestamping would not be significantly different from that of locking, since both activities basically involve table manipulation and updates.

Taking these numbers as reference, this type of processing overhead of concurrency control in general would not amount to unacceptable load on the computer system. However, this overhead does tend to grow as the required throughput of transactions and transaction sizes increase, since both of these increases would cause lock tables, etc., to grow in size and complexity, making each visit to these tables more time-consuming. In addition, as pointed out in <FriedmanSO>, practical experiences with DBMS's such as IBM's IMS show that performance of the system tables (e.g., lock tables) associated with the concurrency control facility (or the 'program isolation facility' in IMS's terms) has important bearings on the overall performance of the DBMS and should be taken into consideration when designing transactions, especially transactions that involve many granule accesses and updates. For example, existence of long transactions, if not carefully handled, often severely degrades performance of unrelated short transactions through interference of the concurrency control facility itself.

However, a potentially more serious problem of concurrency control overhead is the fact that concurrency control has to be performed in a critical section in which no other processes can interrupt. This means that if a transaction is currently manipulating a lock table, no other transactions will be allowed to perform locking and unlocking involving the same lock table until the former is finished. It is this contention for the right to enter a critical section for concurrency control purposes that may prove to be a serious limitation in building a

multi-processor system with a high degree of multiprocessing. For example, if concurrency control consumes about 10 percent of the instruction execution time in a particular critical section per transaction, then the maximum number of effective processors a multiprocessor transaction processing system may have is 10. While this is not a problem when the degree of multiprocessing required is low, it is certainly an issue in designing large multiprocessor database computers such as the one to be described in Section 9.3. Reducing visit frequencies to these critical sections, which means reducing frequencies of setting locks or timestamping, would have a significant impact on designing multiprocessor database computers.

1.2.2.2 IMPACT OF LEVEL OF CONCURRENCY ON DBMS PERFORMANCE: TWO EMPIRICAL CASES

In addition to the processing overhead and critical section limitations, a concurrency control method that results in high volume of transaction blocking and therefore limits the level of concurrency achievable in the system will inevitably cause computing resources to be under-utilized while throughput and response times of the system suffer. In this section, we provide a synopsis of two cases where the level of concurrency in the systems has proven to be the limiting factor of performance of the systems. We will first clarify the notion of level of concurrency through a simple example.

Consider the following interleaved schedule of read and write steps from three transactions t_0, t_1, and t_2:

$W_0(b)\ R_1(a)\ R_2(b)\ W_1(b)\ W_2(c)$

where $R_1(a)$ refers to a request from transaction t_1 to read data element a, and $W_1(b)$ to write data element b, and so on. Suppose the basic timestamping algorithm is used and that the timestamp of t_1 is smaller than that of t_2. Then t_1 will be denied right to write data element b (since at that time b is stamped with a t_2 read timestamp which is greater than that of t_1's), and forced to abort. On the other hand, if this interleaved sequence of access requests is received by a two phase locking facility, t_1's request to put a write lock on b would have to be blocked till t_2 finishes and releases its read lock on b. However, an examination of the schedule reveals the fact that the schedule is serializable and is equivalent to a serialized schedule in which t_1 is executed after t_2. Since the timestamp algorithm is designed to synchronize concurrent transactions by timestamp order, any serializable schedule that violates the timestamp order is ruled out. Similar arguments prevail for the locking protocol.

Blocking or aborting transactions cause the level of concurrency attainable in the system to suffer, which in turn degrades system performance. As shown in the above example, not all interferences from a concurrency control algorithm through blocking or aborting are necessary. Therefore, in <Papadimitriou82>, optimality of a concurrency control algorithm is defined as the degree to which the algorithm allows

for 'serializable input schedules' to proceed without being interrupted. Due to issues of implementability, no known concurrency control methods would allow all serializable schedules to proceed without interference. However, optimization of currently available methods through transaction analysis, such as in the case of the hierarchical database decomposition approach reported in this thesis, can increase the level of concurrency of a DBMS.

Now we use two cases to illustrate the effect of the level of concurrency on system performance.

1.2.2.3 BENCHMARKING A DATABASE APPLICATION

This case was reported by the BDM Corporation <BDM>. In 1982, the BDM corporation, a general consulting firm based in Washington D.C., was contracted to develop a document control system for the Department of Defense, called The Technical Information Management Library and Document Control System. The system was developed on the ORACLE Database Management System, version 2, a commercially availble DBMS released in 1981. The design of the ORACLE DBMS received much influence from System R, an experimental relational database management system developed at IBM Research Lab.

Two observations were made by the BDM development team after performing benchmark testing for the document control system:

(1) In the first set of benchmark tests, two sets of transactions were run against the system. The first set consists of only read-only transactions, and were run without any interference from the concurrency control facility. The second set of transactions are similar in nature to the first except that they are update transactions and therefore requires 3 times as many I/O as the set of read-only transactions. The result of the benchmark is that running the first set of transactions attained a throughput 10 times as much as that of the second set. It was determined that the portion of the discrepency in performance not explainable by the differences in I/O requirements is to be attributed to the concurrency control activity which has apparently limited the level of concurrency attainable by the computer system and severely degraded system throughput. Assuming that each transaction in the second set requires 3 times as much CPU and I/O as that of the first set and therefore can be considered as equivalent to 3 transactions in the first set, one draws the conclusion that for this particular application and set of transactions, turnig off interference from the concurrency control facility may result in a 330 percent increase in throughput. Conversely, concurrency control is responsible for a 75 percent degradation in throughput.

(2) The above benchmark tests do not involve conversational transactions. A conversational transaction is a transaction that involves operator think time. For example, if a user inputs

'Begin Transaction' and requests certain processing be done, then waits for a certain amount of time ('think time') to elapse before inputting the rest of the processing request for that transaction before issuing the 'End Transaction' instruction, this user is executing a conversational transaction. The second set of benchmark tests performed by BDM is a comparison between performance of conversational update transactions and non-conversational update transactions. At the development site, a benchmark of a certain type of update transactions was performed and was reported to have attained a throughput rate of 10 transactions per minute. This test case did not take operator think time into consideration. However, when the system was installed at the user site and was tested with the same type of update transactions with operator think time involved in the execution of these transactions, the throughput suffered dramatically and dropped to 1 transaction per minute, with a response time of 3 minutes per transaction. A diagnosis of the cause of this degradation indicated that interference from the concurrency control facility made it impractical for the system to run more than 3 transactions at a time, severely limiting the use of the potential capacity of the system.

1.2.2.4 PREDICTING PERFORMANCE OF A HIGH-THROUGHPUT DBMS APPLICATION

This case was reported by BGS Systems, a consulting firm based in Massachusetts that specializes in computer system performance evaluation. BGS was requested in 1982 to conduct a study using analytical modeling tools to predict the performance of a high-throughput (approx. 11 transactions per second at peak load) database application implemented on IBM's IMS system, for a client who wishes to remain anonymous. This synopsis of the result of this study is based on a report furnished to the author by BGS Systems <BGS>.

The study began with an analysis that quantified the effect of competition between different IMS processing regions (i.e., transactions) for certain highly-used database records. In particular, it documented the fact that, even through the computer system can theoretically allow a much larger number of transactions to reside in the system, the effective level of multiprogramming attainable by the current application system is about 3, beyond which blocking due to concurrency control would eliminate the benefit of a higher multiprogramming level. This observation is captured in the following two figures extracted from the report.

In Figure 2, predicted system response times in minutes are plotted against throughput load of the system, *if no concurrency control interference existed.* It can be seen that the response time increases very slowly as throughput load increases, signifying that the computer system resources such as CPU and I/O devices are not heavily utilized. In con-

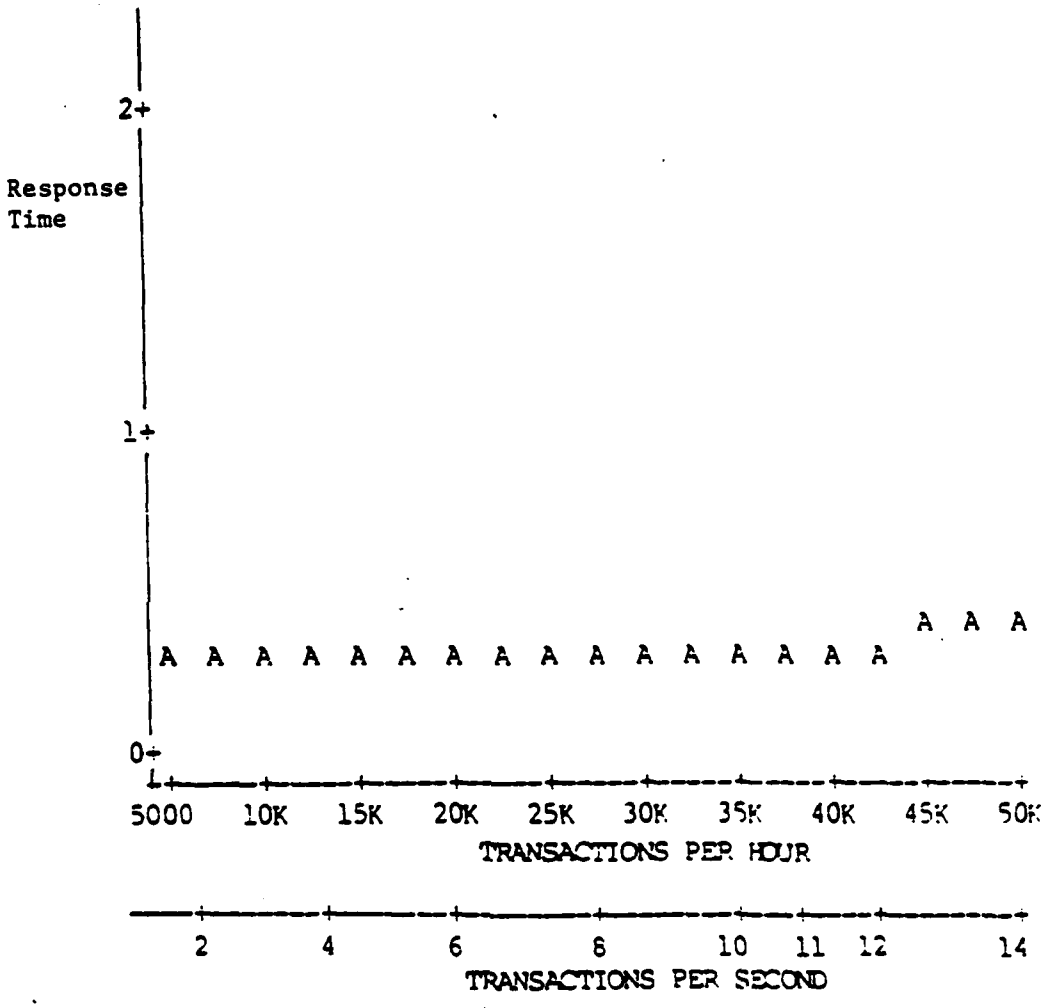


Figure 2. System Response Time Assuming No Blocking Existed

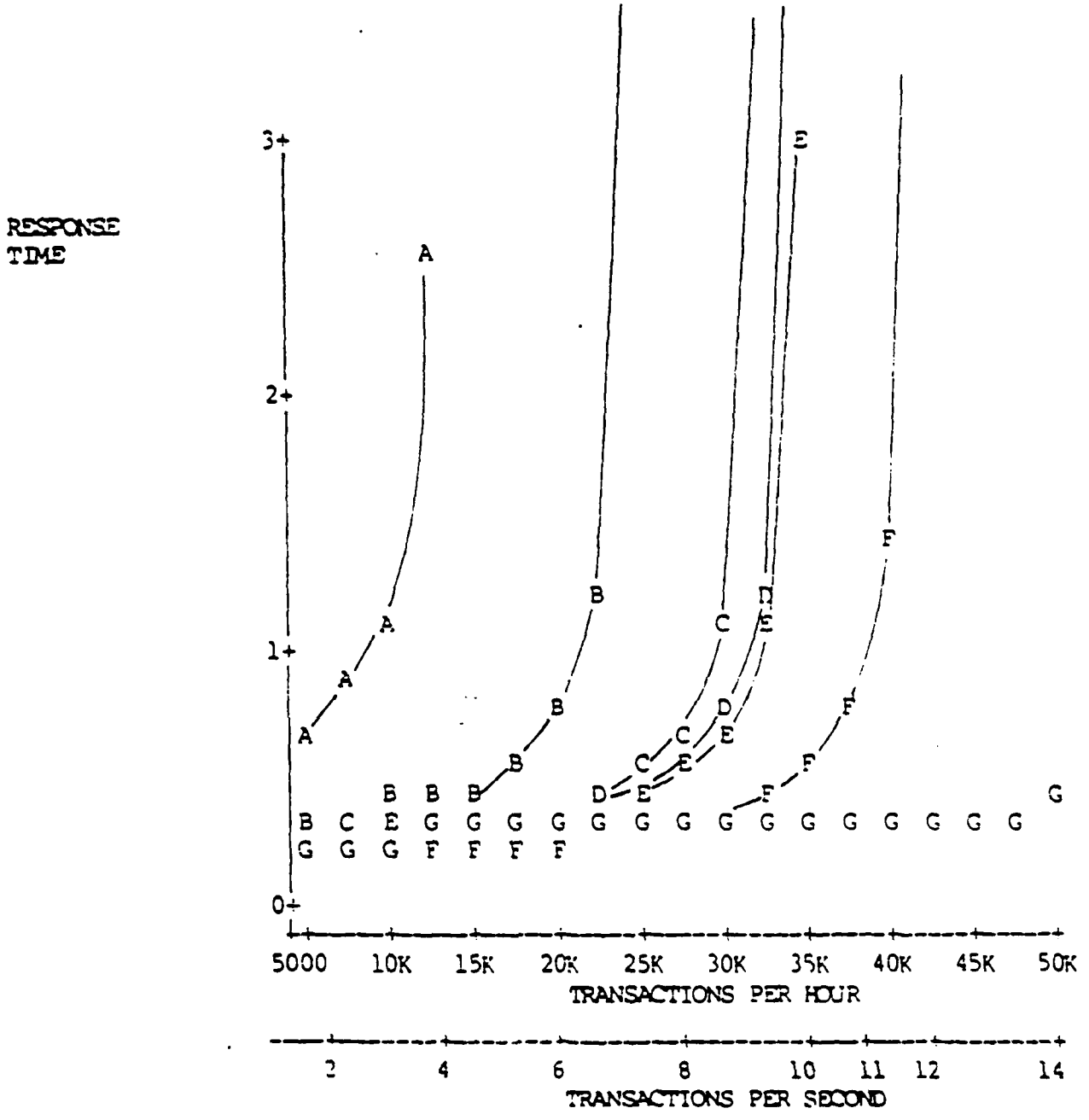


Figure 3. System Response Time With Blocking Effect Included

trast, Figure 3 presents the result of the system performance after introducing concurrecy control blocking effect into the current application. The A-curve represents the response time of the same set of transactions on the same computer system as shown in the previous figure. However, the interference of the blocking effect from the concurrency control facility of the DBMS produced a steep response time curve this time, and, since an average response time in excess of 2 minutes are considered unacceptable by the client, the graph clearly states that the current system can not perform satisfactorily. Curves B through F documents predicted performance under various improvements done to the computer system, including such measures as introducing faster disks and better arrangement of data on the I/O devices. However, it was concluded that, while with these tunings the current system might produce adequate results, it probably would not be able to handle further increases in load that are expected to take place. The study recommended a redesign of the application system to better handle the concurrency control issue.

1.2.2.5 SUMMARY

The above two cases are included in this chapter to provide empirical evidences that the concept of system limitation due to database concurrency control is an important one and is expected to gain attention as databases are more widely installed and loads on these systems increase.

1.2.3 AN EXAMPLE ILLUSTRATING THE SCOPE OF CURRENT RESEARCH

Consider an inventory database application of a retail business with a database shown in Figure 4. There are several types of transactions that operate on this database. A type 1 transaction inserts a sales record into the database when the event of a sales occurs. A type 2 transaction inserts a stock-arrival record into the database when the event of a stock arrival occurs. A type 3 transaction is generated periodically for each item in the inventory to compute the current inventory level of that item. This transaction visits the sales and stock-arrival records to compute the net change ever since the last computation of the inventory level of that item. A new inventory level for that item is then posted in the inventory record.

To control these transactions using two-phase locking, a type 3 transaction would have set read locks on every read access it has generated to the sales and stock-arrival records, including records serving as access paths such as index records. Using the timestamp ordering approch, the type 3 transaction would have left read timestamp for every such record it has read. What we are interested in here is whether there is a concurrency control method that would eliminate the need for leaving read timestamps (or setting read locks) when a type 3 transaction accesses the sales and the stock-arrival records. An analysis shows that, because those transactions that update the sales and the stock-arrival records do not access the portion of the database a type 3

- DATABASE

  Inventory

Item#	Qty-on-hand

  Sales

Item#	Qty-sold

  Stock-Arrival

Item#	Qty-received	Order#

  Stock-On-Order

Item#	Order#	Qty

- TRANSACTIONS

  TYPE 1: POST Sales

  TYPE 2: POST Stock-Arrivals

  TYPE 3: EXAMINE Sales AND Stock-Arrivals TO ADJUST Qty-on-hand

  TYPE 4: EXAMINE Stock-Arrivals AND Qty-on-hand TO MAKE REORDER DECISIONS

Figure 4. An example inventory database.

transaction would update, the above objective can be achieved as follows: if one could produce a snapshot of the sales and the stock-arrival records for a type 3 transaction, say, t, when t is initiated, and have t operate on the snapshot rather than the original records, then, without compromising serializability, t would not have to set read locks or leave read timestamps on these records. This means that concurrent updaters of these records (i.e., type 1 and type 2 transactions) can proceed without being interfered by t.

Let us introduce yet another type of transactions. Suppose there are type 4 transactions which are also generated periodically to check for the need of reordering certain stock items. This type of transaction reads the stock-arrival records and adjust the stock-on-order records (by setting the arrival-date field of such records.) It then computes a gross inventory level by summing the current-inventory-level and the quantities indicated by the non-arrived stock-on-order records. Based on this gross inventory level a decision is made as to whether to reorder this item. If it decides to do so, an order request is printed and a stock-on-order record is generated and inserted.

It is noted that a type 4 transaction reads, but does not update, the stock-arrival and the inventory records. Therefore it is desirable if one can generalize the above observation on the type 3 transaction to apply to the type 4 transactions such that the read accesses to the stock-arrival and inventory records by a type 4 transaction can proceed

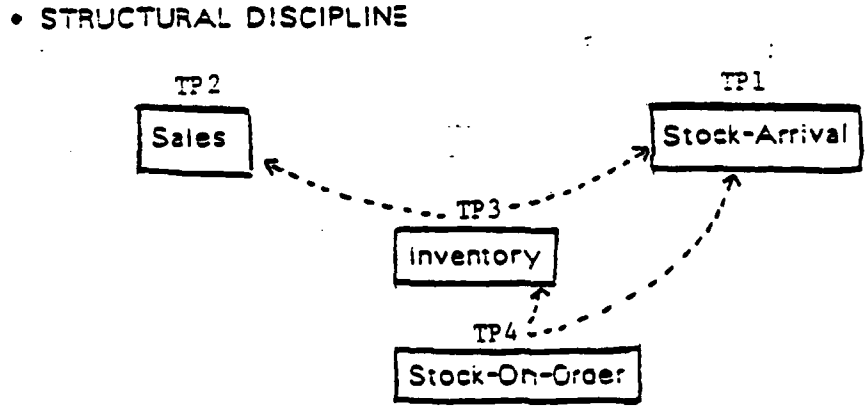


Figure 5. The Structure of the Example Application

without inducing interference with the other three types of transactions.

This example depicts a database system whose applications are structured in a special way as shown in Figure 5. In the figure it is shown that a transaction type can be paired with a particular portion of the database it updates and has read accesses (dotted arrows) to other portions of the database. In fact, the above case can be generalized further by, for example, adding to it transaction types that read from the reordering records and the stock-arrival records to generate supplier profile records in the database. As the list goes on, a hierarchy of transaction types is formed such that at each level of the hierarchy the transaction only reads from, but does not, or rarely, write into data written by transaction types of an earlier level. This structure presents a definite opportunity for concurrency control algorithms to explore.

1.2.4 RESEARCH GOALS AND ACCOMPLISHMENTS

Most concurrency control algorithms, for simplicity, choose to ignore the above opprotunity for reducing synchronization overhead, and complies strictly with the assumption that all transactions may read and write any part of the database and therefore every access has to be controlled. The goal of the current research is to demonstrate that a *systematic exploitation* of the read-write pattern of transactions can be

effectively used to achieve the objective of reducing concurrency control overhead.

In this thesis, we will demonstrate the potential benefit of incorporating into a concurrency control algorithm the knowledge of special structures of the applications. Sepcifically, we accomplish the following tasks:

(1) Formalize the concept of 'hierarchically organized database applications' and identify types and properties of such organizations.

(2) Develop a timestamp-based concurrency control algorithm, called the Hierarchical Timestamp Algorithm (HTS), that is capable of taking advantages of the existence of such organizations in database systems. Specifically, we achieve the goal of allowing certain read acesses requested by update transactions to proceed without ever having to set read timestamps or having to create the danger of causing other concurrent transactions to abort. In other words, the algorithm allows certain update transactions to proceed in a way that does not interfere at all with certain other concurrent update transactions. In addition, a protocol for synchronizing the read-only transactions in a hierarchically organized database system is also developed which allows the read-only transaction to proceed in a way that does not interfere at all with any concurrent update transaction.

The algorithm, consisting of a total of three types of protocols for synchronizing different types of accesses from update transactions, can be considered as a generalized timestamp algorithm parameterized by a hierarchical structure of the applications. This algorithm degenerates to a conventional timestamp algorithm (therefore consisting of only one relevant protocol) when such a hierarchical structure is not recognized.

(3) Propose an implementation scheme for the hierarchical timestamping algorithm which demonstrates the benefit of the above algorithm in concrete terms. In designing the implementation, a methodology is also developed for analyzing and laying out system modules and shared data structures where concurrency is of concern. This methodology is believed to have bearings on concurrent system design in general.

(4) Formulate as an optimization problem the task of recognizing a favorable hierarchical structure given a database application system. The problem, unfortunately, is proven to be NP-hard, and a heuristic algorithm is proposed for solving it.

(5) Apply the concept of the hierarchical database organization approach to concurrency control to three different problem areas and demonstrate the efficacy of incorporating this concept into solving the concurrency control problems in database management systems.

It is important to stress that the thrust of the thesis research is more than proposing a new concurrency control algorithm. It demonstrates the potential benefit of exploiting the knowledge and the structure of the application systems to implement more efficient and more tailored concurrency control mechanism. It also points out an area of research that has not been fully examined, namely, *how to design transactions so that concurrency control tasks can be simplified without compromising the managerial requirements on the data.* It is believed that transaction design with the concurrency control problems in mind could produce a structure of applications that is easier and less costly for concurrency control to be implemented.

1.3 THE STRUCTURE OF THE THESIS

The thesis is composed of a total of ten chapters. In the first chapter, the problem of database concurrency control is introduced and the motivation and the scope of the thesis research is discussed. It also summarizes the goals and the accomplishments of this thesis research.

In Chapter two, a literature overview is presented to bring out relevant recent research developments in the database concurrency control area. It provides a perspective as to how this thesis research is related to recent trends in concurrency control research. This chapter

also contains a review of important formalisms involved in the concurrency control problem so as to provide the background for understanding the proofs to be presented in the subsequent chapters.

In Chapter three, the concept of the hierarchical organization of database applications is formalized. It classifies the tasks of algorithm development in such organizations into three: protocols for controlling update transactions under non-cyclic database partition, protocols for controlling update transactions under cyclic database partition, and protocol for controlling read-only transactions where udpate transactions obey the above protocols. These three tasks are separately described in Chapters four to six. In each of these chapters, the basic definitions and the protocols are first described with illustrating examples. The proofs of correctness then follow. These protocols collectively form the hierarchical timestamp concurrency control algorithm.

In Chapter seven, a scheme is proposed for implementing the hierarchical timestamp protocols developed in the previous chapters. This scheme addresses the problems of timestamp computation and mangement, multi-version database implementation and the problem of garbage collection.

In Chapter eight, we examine the problem of how to identify a favorable hierarchical database organization that can be used by the hierar-

chical timestamp algorithm. The complexity of this problem is analyzed and a heuristic algorithm for solving it is proposed.

In Chapter nine, we discuss the efficacy issue of the hierarchical timestamp approach to database concurrency control. Three problem areas (the B-tree access method, a banking application and a database computer application) are explored.

Finally, in Chapter ten, we summarize the thesis and discuss future research directions.

2.0 LITERATURE REVIEW

This chapter is composed of two parts. The first section is an overview of the recent works and trends in concurrency control research to provide a perspective as to how the work reported in this thesis is related to other works in the field. The second section reviews basic concepts and formalisms in the concurency control theorey to provide a background for the theoretic developement to follow in the subsequent chapters.

2.1 OVERVIEW OF RELEVANT RECENT RESEARCH IN DATABASE CONCURRENCY CONTROL

Concurrency control in a centralized or a distributed database system has been an active research area. The concept of database consistency has been formally analyzed in <Eswaran76, Gray76>, in which set-theoretic notions are used to formulate the concept. A consistent schedule of concurrent transactions has been defined to be one which is equivalent to a serialized schedule. Two-phase locking has been proposed in their papers as a mechanism which preserves serializability. This notion of serializability has been further explored in <Bernstein79> and <Papadimitriou79>.

2.1.1 ALGORITHMS, METHODS AND OTHER ISSUES

Algorithms for database concurrency control abound in the literature. The distributed database present a challenge for the consistency problem which has encouraged development of many new algorithms. <For example, Ellis77, Lamport78, Rosenkrantz78, Thomas79, Bernstein81.> A survey and comparison of theories and algorithms of concurrency control can be found in <Bernstein81>.

Most algorithms are considered variations, extensions and/or combinations of the two basic techniques for concurrency control - two-phase locking and time stamp ordering. The two-phase locking algorithm ensures consistency by imposing a partial order on all transactions based on their lock points. (A lock point of a transaction is the point in time when the locking phase of the transaction reaches its peak.) The timestamp ordering algorithm, on the other hand, ensures consistency by imposing a partial order on all transactions based on the initiation times of the transactions. The latter is sometimes referred to as the optimistic concurrency control method <Ullman82> because it allows transactions to proceed without locking the data elements in hope that conflicts would not occur. It, however, timestamps the data elements with the timestamp assigned to the transaction (i.e., usualy the initiation time of the transaction) that accesses the data elements in order to detect conflicts when they do occur and resolve them through trans-

action aborts. These two techniques, two-phase locking and timestamp ordering, have been used as the basis for further explorations.

Another optimistic alternative to two-phase locking for concurrency control is the validation method <Kung81, Ceri82>. Like the timestamp algorithm, the validation method allows the transaction to proceed relatively freely without setting locks on data elements accessed. Unlike the timestamping method, however, the algorithm does nothing (i.e., not even timestamp data elements) during the execution of the transactions. Instead, at the end of the transaction execution, the transaction enters a 'validation phase' during which the set of the data elements the transaction has accessed is intersected with the access sets of all other concurrently committed transactions. The transaction is validated if the intersection is empty, and is aborted if it is not. Obviously, this method rests on the assumption that conflicts rarely occurs, and therefore virtually all transactions would pass the validation phase. Other variations of the validation method have also been discussed <Bernstein82b>.

Other issues concerning concurrency control algoirthms in general are the data granularity and crash recovery. In <Gray75>, <Korth82> and <Carey83> the issues involved in granularity hierarchies are discussed, while in <Ries77> and <Ries79> the effect of the choice of data granules on the performance of the concurrency control algorithms is addressed. In <Gray78>, the two-phase commit policy is discussed which addresses

the issue of handling a system crash during a transaction's write phase. In order not to have the effects of such such transactions cascade in an undesirable fashion, the two-phase commit policy is used to effect 'atomic commit', which stipulates that in-place database updates should not take place until a transaction has forced all its upates to a secured log.

2.1.2 MULTI-VERSION METHODS

One recent development in concurrency control algorithms centers around the identification of techniques that increase the level of concurrency and/or reduce synchronization overhead, at the same time preserving the correctness of the algorithm. One approach is the use of a multi-version database. It's been observed that keeping multiple versions of database elements will improve concurrency of the database <Papadimitriou82>. In Papadimitriou's paper, it is shown that there exists an infinite hierarchy of multi-version serializability, and proven that the more versions a DBMS keeps, the higher is the level of concurrency it may achieve.

The concept of a timestamp-based multi-version database system was first proposed in <Reed78>. It is a general scheme in which the identifier of a data element consists of two components: the name of the data and the version of the data. In Reed's scheme, retrieval of an arbitrary time slice of the database is allowed.

A more limited multi-version concept was developed in <Bayer80>. In his scheme, one previous version of a data element, which has been saved for recovery purposes when the data element is going through changes made by uncommitted transactions, is utilized to allow read accesses to proceed without having to wait for the commitment of the update transaction. However, read locks still have to be set by the read requests and there is the additional overhead of explicitly maintaining a transaction dependency graph. An extension of this method to a distributed database is proposed in <Bayer80b>. In <Viemont82>, an interesting method for concurrency control is devised which also makes use of this one extra copy of data elements to synchronize transactions by order of commit time. In essance his technique is one which blends timestamp ordering and two-phase locking in one and chooses to switch to one or the other at the most opportune time so as to increase the level of concurrency. In <Stearns81, Chan82> the one-previous-version method was extended to accommodate multiple previous versions (but does not allow for access of an arbitrary time slice of the database from a user.) Stearns' method extends the method propsoed in <Bayer80> to exploit the benefit of the existence of multiple previous versions. Read locks, however, are still needed for all transactions. Chan's method is also based on two-phase locking but allows the read-only transaction to receive special treatment - they do not have to set read locks. Extending this method to a distributed database is presented in <Chan82b>. In <DuBourdieu82> a method similar to Chan's is also discussed. In

<Garcia-Molina82>, a framework of strategies for processing read-only transactions is presented.

The above list of research bears resemblance to the research to be reported here. However, our technique is one which is timestamp based and strives to reduce the need for leaving read timestamps for not just read-only transactions, but update transactions as well.

2.1.3 EXPLOITING KNOWLEDGE OF TRANSACTIONS

Another approach to reducing synchronization overhead is to use the method of transaction analysis to obtain a priori the knowledge of the nature of the transactions to be run in the system. Two notable examples are the conflict analysis <Bernstein80b> and the hierarchical locking protocol <Silberschatz80, Kedem80>.

In the research on concurrency control for SDD-1, conflict analysis was proposed which exploits a priori knowledge of the nature of the transactions to be run in the system. (To some extent, some of the research listed in the previous paragraphs concerning providing special protocols for read-only transactions falls into this category too, as it exploits the knowledge, albeit a limited one, of the nature of the transactions, namely, whether they are read-only or not.) The basic idea behind the SDD-1 approach is 'synchronizing only when necessary'. It synchronizes a transaction in one class with only those transactions

in classes that are in conflict with that transaction. If only two transactions are run concurrently and they belong to classes that, via transaction analysis performed before run time, do not conflict, then there is no need to incur control overhead. However, if two transactions are in conflict, then timestamps plus intra-class serialized pipelining are used to ensure consistency.

The approach reported in the present paper is different from that of SDD-1 because it is not oriented towards distributed database systems, and, because of the special structure of applications that our approach exploits, together with the fact that the multiple version technique is employed, the protocols are much less restricted. These new protocols are therefore more practical to implement.

The hierarchical locking protocol, sometimes referred to as the tree protocol, also exploits special knowledge of the nature of the transactions. Specifically, it assumes a priori knowedge that the sequence of accesses to the data elements exhibits a tree pattern. For example, let data elements x and y in the database be accessed by a transaction only if the transaction has previously accessed another data element z, then one may consider z as the 'parent' of x and y and the access path that a transaction follows is always from the parent data elements to the child data elements. In general, if all data elements in the database can be structured as a tree and the access path to these elements followed by transactions are always congruent to the tree structure,

then a non-two-phase locking protocol can be used to synchronize the accesses. The tree protocol is correct, but it is not two-phase and is capable of allowing a higher level of concurrency than the two-phase locking algorithm. Generalizing the tree protocol to DAG (directed acyclic graph) has also been attempted. This concept of exploiting structural knowledge of the database applications to identify better concurrency control methods is further discussed in <Kung79>, where the notion of 'optimality of concurrency control algorithm' is explored.

While the research reported in this thesis also uses acyclic graphs as tools for analyzing transactions and organizing groups of data elements (hence the term 'Hierarchical Timestamp Algorithm'), it bears little relationship with the hierarchical locking protocol developed by Kedem and Silberschatz. The hierarchical locking protocol is concerned with the knowledge of the natural *sequence* of the data requests from the transactions (e.g., first z, then x and y), and attemps to achieve early releases of locked elements by exploiting this knowledge. The HTS algorithm is not concerned about the sequence of accesses but the modes of accesses from groups of similar transactions (e.g., always read z but never write z), and exploits this knowledge to achieve the effect of not having to leave traces for reading data elements. Therefore these two approaches, while sharing the philosophy of 'knowledge exploitation' and the tool of acyclic graphs, are very different in substance.

In the following section, we will review the basic formalisms in database concurrency control and pave the way for formally developing the hierarchical timestamp algorithm in the subsequent chapters.

2.2 BASIC CONCEPTS OF MULTI-VERSION CONSISTENCY

In this section, we review the basic concepts and formalisms in database concurrency control. The concurrency control theory centers around the notion of 'correctness' of schedules. A *schedule* is a sequence of steps such as read, write or read-and-write of data elements in the database. An *input schedule* is a sequence of steps representing requests for actions on data elements submitted to the concurrency control (CC) facility by the transactions. An *output schedule* is a sequence of steps allowed by the concurrency control facility, which is basically a permutation of the input schedule. The purpose of the CC facility is to take an arbitrarily interleaved input schedule and produce an output schedule containing the same steps but in a sequence that is *correct*.

2.2.1 TRANSACTION PROCESSING MODELS

To prove that a concurrency control algorithm is correct, it is necessary that a precise definition of correct schedules be given. As discussed before, serializability is a common criterion, and is defined as follows:

Definition. A schedule S is ***serializable*** if there exists an equivalent schedule S_s where all transactions in S_s are serialized. (i.e., no steps of one transaction are interleaved with steps from another transaction.)

Now we must give the precise meaning of 'schedule equivalence'. Intuitively, two schedules of a set of transactions are equivalent if their 'effects' on the database are equivalent. Since it is only realistic to assume that the CC facility is blind to the intent of a transaction's use of a data element, the common notions of equivalence can only be based on the assumption that every data element retrieved by a transaction would have an impact on data elements later written by the transaction, and every write changes the value of the data element.

However, the above concensus does not eliminate the existence of the different ways in which the meaning of a 'step' in a schedule is interpreted. As a result, there are various ***transaction processing models*** each with its own notion of a 'step' in a schedule. The simplest model is the action model in which it is assumed that every step in a schedule is both a read and a write of the requested data element <Stearns76>. A more general model differentiates a read-only step from a read-write step, and is sometimes called a conflict model <Bernstein79>. The most general model is the read-only/write-only model, which does away with the assumption implied in the conflict model that a write must entail a previous read. <Papadimitriou79>. The notion of schedule equivalence

therefore must depend on the underlying assumption about what each step in a schedule means.

In addition to the differences due to transaction processing models, schedule equivalence also depends on whether it is meant to be 'final-state equivalence' <Papadimitriou79> or 'view equivalence' <Rosenkrantz82>. The former stipulates that two schedules are equivalent if the final database states are equivalent. Under this assumption, those transactions that do not generate any effect in the final database state (e.g., read-only transactions) can be considered redundant, or 'dead', and would not in any way affect the decision as to whether the schedule is serializable. The latter is concerned with whether every transaction sees the same database state in the two schedules rather than whether the final database state is the same.

The differences in defining schedule equivalence results in different technical definitions for serializability, which in turn results in different algorithms for testing for serializability. For example, testing serializability of a schedule under the action model amounts to detecting cycles in a simple directed graph, for which a polynomial time algorithm exists, while testing that under the read-only/write-only model amounts to detecting cycles in a polygraph, which is NP-complete.

In sum, to prove that a concurrency control mechanism is corret, one must state the assumptions underlying the transaction processing model

and the precise meanning of schedule equivalence. With this background, we now present the basic concepts in multi-version serializability.

2.2.2 MULTI-VERSION SERIALIZABILITY

Two definitions for mutli-version serializability have recently been proposed. Both definitions are based on the most general transaction processing model, namely, the read-only/write-only model, but differ in the definitions of the read steps in a schedule. The difference lies with whether the version number of the data element of a read step is specified.

In <Bernstein82>, a multi-version (mv) schedule is composed of read steps in the form of $r_j(d^i)$ and write steps in the form of $w_i(d^i)$, where $r_j(d^i)$ is interpreted as 'transaction j reads the version i of data element d' and $w_i(d^i)$ is interpreted as 'transaction i creates the version i of data element d.' Any feasible mv schedule is subject to the constraint that if $r_j(d^i)$ is in the schedule, then $w_i(d^i)$ must be before $r_j(d^i)$ in the schedule. This constraint merely says that a version cannot be read until it has been created. Under this definition, the version that a read step is reading is determined in the schedule. To test for serializability of a mv schedule defined in this way, one must find out if there exists a *version order*, such that, given this version order denoted as <<, the graph defined below is acyclic: (There are notational

differences in the definition given here and that given in <Bernstein82>)

Definition. A transaction dependency graph of a mv schedule S and a version order <<, denoted as TG(S,<<), is a digraph where the nodes are the transactions in S and the arcs, i -> j, representing 'transaction i depends on j', are assigned according to the following rules:

i -> j iff

(1) $w_j(d^j)$ and $r_i(d^j)$ are in S, or

(2) $r_j(d^k)$ and $w_i(d^i)$ are in S and k << i, or

(3) $w_j(d^j)$ and $r_k(d^i)$ are in S and j << i.

Intuitively, this means that a transaction i is said to depend on (i.e., must follow) transaction j if i reads a version of a data element created by j, or i writes over (i.e., creates a subsequent version) a version of a data element read by j, or i writes over a version of a data element created by j and the version created by i is read by any transaction. Assuming a read-only/write-only data model, the following theorem is given in <Bernstein82>:

Theorem. A mv schedule S is serializable iff there exists a version order << such that TG(S,<<) is acyclic.

It is shown that, under this definition, testing for serializability given a mv schedule is NP-complete. Intuitively, the complexity of the

testing algorithm stems from the combinatorial explosion of the number of possible version orders that the algorithm must examine. However, if a version order is given, then the problem is reduced to one which tests for cycles in a simple graph, which is not a hard problem.

In <Papadimitriou82>, a mv schdules is defined to be composed of read steps in the form of $r_j(d)$ and write steps in the form of $w_i(d^i)$, where $r_j(d)$ denotes 'transaction j reads an (unspecified) version of d' and $w_i(d^i)$ has the usual meaning. In other words, the mv schedule in this definition retains the flexibility of assigning versions to be read, so long as the assignment satisfies the constraint that a version must be created before it is read. With this additional flexibility, testing for serializability of a schedule must also take into consideration all the possible ways of assigning versions to the read steps. Therefore, a mv schedule is serializable if and only if there exists an *interpretation* I of the schedule, i.e., an assignment of versions to read steps in S, and a version order <<, such that the transaction dependency graph TG(S,I,<<) is acyclic. Intuitively, the term 'an interpreted schedule' is equivalent to the term 'a schedule' in the previous definition. It is not difficult to see that testing for serializability of a schedule under this second definition is also NP-hard, while testing for serializability of an interpreted schedule given a version order is equivalent to testing for existence of cycles in a simple directed graph.

It can be concluded that, while testing for serializability of a multi-version schedule is a hard problem, it is quite straight-forward if the mv schedule is interpreted (i.e., version numbers have been assigned in the read and write steps) and a version order is determined. For our purpose, fortunately, the schedules that must be tested for serializability, in order to prove the correctness of the protocols, are interpreted, and the specific version orders are given. We summarize the above discussion by giving the following definitions and a theorem which give the definitive word on how to prove correctness of our mutli-version concurrency control algorithm to be developed in the subsequent chapters.

Definition. A ***schedule*** S of a set of transactions T is a sequence of steps, each of which is denoted as a tuple of the form $a_i(d^j)$. a_i refers to an 'action' on behalf of a transaction t_i. The action can be read (r) or write (w). d^j is a version of a data granule, where d indicates the data granule and j indicates the version. If the action is write, then the version of the data granule included in the step is created by the transaction. If the action is read, then the transaction reads the version of the data granule indicated in the tuple.

An example of a schedule is $w_1(d^1)$, $r_2(d^1)$, $w_2(d^j)$, $r_3(d^j)$.

Definition. Assume that a version order, denoted as <<, is given. A version j of a data element d is the ***predecessor*** of a version k of d if j << k and there exists no version i of d such that j << i << k.

Definition. A ***transaction dependency graph*** of a schedule S is a digraph, denoted as TG(S), where the nodes are the transactions in S and the arcs, representing ***direct dependencies*** between transactions, exist according to the following rules:

$t_2 \rightarrow t_1 \in A$ iff

(1) $w_1(d^j)$ and $r_2(d^j)$ are in S for some d^j, or

(2) $r_1(d^j)$ and $w_2(d^k)$ are in S for some d^j, d^k where d^j is the predecessor of d^k, or

(3) $w_1(d^j)$, $w_2(d^k)$ and $r_3(d^k)$ are in S and d^j is a predecessor of d^k.

In other words, the transaction dependency graph represents a relation $\rightarrow$ (i.e., 'depends on', or 'follows') of transactions such that $t_2 \rightarrow t_1$ if t_2 reads a version of a data granule created by t_1 or if t_2 creates a version of a data granule whose predecessor has been read by t_1, or t_2 creates a version of a data granule whose predecessor was created by t_1 and the version created by t_2 is eventually read by some transaction in S.

The following theorem follows from the discussion given previously:

Theorem. Given a version order <<, a schedule S is serializable iff TG(S) is acyclic.

3.0 HIERARCHICAL DATABASE DECOMPOSITION - DEFINITIONS AND PROPERTIES

As discussed in Chapter one, the hierarchical timestamp algoirthm for concurrency control is devised to explore the opportunity that a set of hierarchically organized applications of a database may present. A hierarchy of applications was briefly described as one in which the transactions belonging to applications at one level of the hierarchy would only read from, but would not, or would rarely, write into data produced by applications of an earlier (i.e., a higher) level. Hierarchically organized applications may offer an opportunity to optimize a concurrency control mechanism because those read accesses from a transaction at one level to data produced by transactions at a higher level may be synchronized using a special, less expensive protocols. In this chapter, we will formalize this concept of 'a hierarchy of applications' and set the stage for presenting the actual algorithms in the subsequent chapters.

We capture the essence of a hierarchy of applications through the definition of a *data segment hierarchy*. A data segment hierarchy, constructed during the database installation phase (i.e., before run time), is basically a partial order of a given set of data segments of the database subject to additional graph-theoretic constraints and constraints related to transaction accesses. In this chapter we will

discuss only *what* constitues (i.e., the formal definition of) a data segment hierarchy given a database segmentation. We will not, however, address the issue of *how* the database segmentation scheme is arrived at and how an optimal data segment hierarchy, if there exist more than one, can be found. The latter issues will be addressed in a later chapter on database decomposition methodology.

3.1 DATABASE PARTITION AND THE DATA SEGMENT GRAPH (DSG)

Let the database be partitioned into a number of *data segments*. We will use the concept of a *data segment graph (DSG)*, constructed by means of transaction analysis, to reflect the accesses by database transactions to the data segments in a database. As will be shown later, the topology of the data segment graph will be used to analyze whether a particular partial order of the data segments constitute a data segment hierarchy relevant to the hierarchical timestamp (HTS) concurrency control scheme.

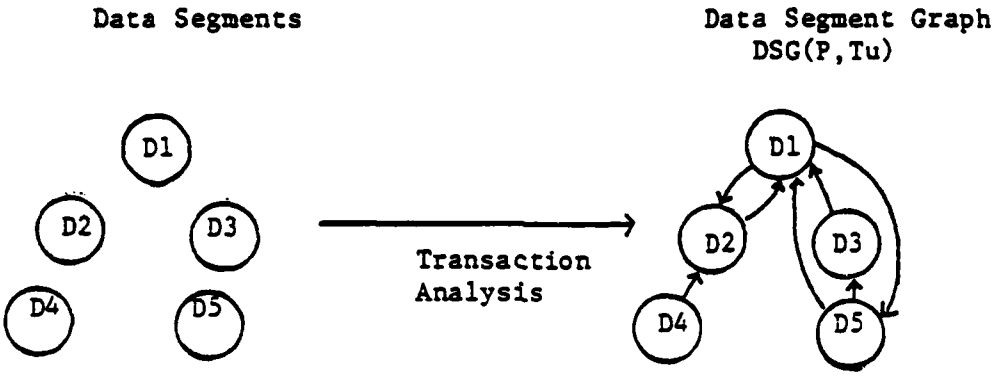
Informally, let a database be partitioned into data segments. A DSG is a digraph with nodes corresponding to the data segments and arcs constructed in such a way that there is an arc from a data segment D_i to another data segment D_j if and only if one can find a potential transaction in the database system that updates data elements in D_i and accesses (i.e., reads or writes) data elements in D_j. In other words,

$D_i \rightarrow D_j$, $i \neq j$, indicates that there exist transactions in the system that would link updates of data elements in D_i to the content of data elements in D_j.

Definition. Let T^U be a set of *update* transactions to be performed on a database D. Let P be a partition of D into ***data segments*** $D_1, \ldots, D_n$. A ***data hierarchy graph*** of P with respect to T^U is a digraph denoted as $DSG(P,T^U)$ with nodes corresponding to the data segments of P and a set of directed arcs joining these nodes such that, for $i \neq j$, $D_i \rightarrow D_j$ iff there exist $t \in T^U$ s.t. $w(t) \cap D_i \neq$ empty and $a(t) \cap D_j \neq$ empty, where t is a transaction, w(t), r(t) and a(t) the write set, the read set and the access set of transaction t. (The access set a(t) is the union of r(t) and w(t).)

Construction of a data segment graph given a set of data segments and update transactions is illustrated in Figure 6. The data segment graph is a tool for capturing the pattern of accesses among transactions to be run in the system. Note that, in our algorithm, there is no need for read-only transactions to participate in this transaction analysis, eliminating the difficulties of pinning down, a priori, the nature of all ad hoc queries.

3.2 THE DATA SEGMENT HIERARCHY (DSH)



where Transaction Analysis is performed on the set of update transactions Tu in the system, and Di → Dj iff (1) i ≠ j and (2) among transactions that write into Di, there exists one that reads from or write into Dj.

Figure 6. Illustration of a data segment graph DSG.

In this section we will give the definition and the properties of the data segment hierarchy. We first briefly introduce the concept of a digraph called a semi-tree. This concept will then be used in the definition of data segment hierarchy. Informally, a semi-tree is a digraph such that, if the directions of the arcs in the graph are ignored, the graph appears to be a spanning tree.

Definition. A ***semi-tree*** is a digraph such that there exists at most one undirected path between any pair of nodes in the graph. Every arc in a semi-tree is called a ***critical arc.***

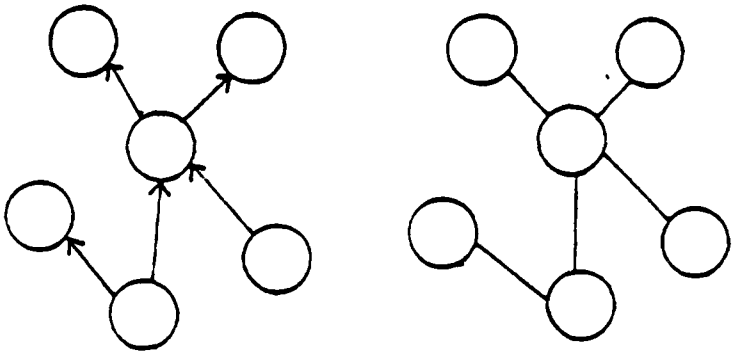
An example of a semi-tree is shown in Figure 7. By deinition, a semi-tree has the property that there exists at most one directed path between any pair of nodes. Now we give the definition of a data segment hierarchy.

Definition. Given a data partition P and a data segment graph DSG(P,T^u), a ***data segment hierarchy***, denoted as DSH(P,T^u), is the transitive reduction of any non-cyclic graph of nodes in DSG(P,T^u) such that

(1) DSH(P,T^u) is a semi-tree, and

(2) If D_i -> D_j is contained in DSG(P,T^u) then either D_i -> D_j or D_j -> D_i is contained in the transitive closure of DSH(P,T^u).

In other words, a data segment hierarchy for a given partition is a partial order of the data segments such that between any pair of data



Semi-tree Spanning tree

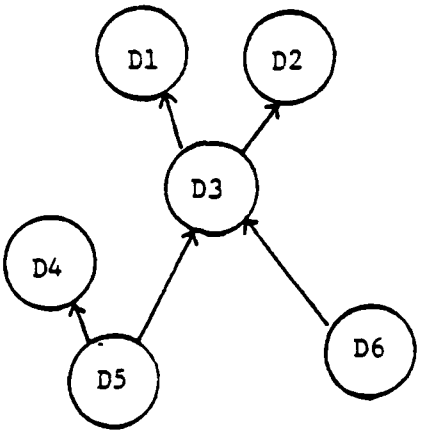


Figure 7. Illustration of a semi-tree and a data segment hierarchy.

segments that are ordered in the partial order there exists only one hierarchical path that leads one data segment to the other. In addition, any pair of data segments that are connected via a directed path in the data segment graph, defined in the previous section, must also be ordered in this partial order. An illustration of a data segment hierarchy is also shown in Figure 7.

The reason why these restrictions are placed in the definition of a data segment hierarchy, i.e., the reason why it is not simply the transitive reduction of any non-cyclic graph, will be made clear after we define the concept of transaction classfication in the next sub-section. Here we address the issue of the existence of DSH given any database partition.

There may be multiple data segment hierarchies that satisfy the above definition given a database partition. However, since any total order of the data segments is a semi-tree, the existence of a data segment hierarchy given a database partition is always guaranteed because any total order that has DSH(P,T^u) as a subset can be used as a data segment hierarchy.

Lemma 0.1. (Existence of DSH) Given a database partition P and a set of update transactions T^u, there exists a data segment hierarchy DSH(P,T^u).

In the remainder of this thesis, the notation DSH* refers to a particular chosen data segment hierarchy. We will say that a data segment D_i is ***higher than*** a data segment D_j, denoted as $D_i \uparrow D_j$, in a data segment hierarchy DSH*, if there exists a path in DSH* from D_j to D_i. In general, we say that data segment D_i and data segment D_j are ***related*** if either $D_i = D_j$ or D_i and D_j are connected by a directed path. We will also denote the path from D_i to D_j in DSH* as CP_i^j (CP standing for Critical Path.)

3.3 TRANSACTION CLASSIFICATION

As discussed before, the purpose of defining a data segment hierarchy for a database is to formally capture the concept of a 'hierarchy of applications'. Therefore the concept of a data segment hierarchy must be paired with a scheme which assigns the transactions in the system to correspond to some levels in the data segment hierarchy. We accomplish this by 'classifying' transactions based on the hierarchical positions of the data segments that the transactions have to write into. We first give the following property.

Property. If there exists a transaction t in T^U such that $D_i \cap w(t) \neq$ empty and $D_j \cap w(t) \neq$ empty and $D_i \neq D_j$, then D_i and D_j are ordered in $DSH(P,T^U)$.

Proof. Follows directly from the definition of the data segment hierarchy.

That is, if a transaction writes into more than one data segment, then all the data segments that it writes into must be connected by a directed path in the data segment hierarchy.

Given a data segment hierarchy DSH*, we will assign update transactions in T^u to the *highest* data segment it writes into. (The fact that a highest data segment exists follows from the above property.) Formally,

Definition. Given a DSH*, a transaction t is *rooted in* a data segment D_i, denoted as $t \in D_i$, iff (1) $w(t) \cap D_i \neq$ empty, and (2) if there exists $D_j \neq D_i$ such that $w(t) \cap D_j \neq$ empty, then $D_i \uparrow\!> D_j$ in DSH*.

As a digression, we can now discuss the motivation behind our definition of DSH. The goal is to build a partial order of the data segments such that we can define a 'rooting scheme' of the transactions such that (1) if a transaction is rooted in D_i and has to access data elements in D_j then D_i and D_j are ordered in the data segment hierarchy (i.e., it is possible to determine whether an access is from a transaction rooted in one data segment to a higher, lower or the same data segment in the segment hierarchy,) and (2) this ordering is established through a unique path in the partial order. These are the basic proper-

ties of a DSH that, as will be shown later, are relied upon by our concurrency control algorithm. The following property formalizes the above discussion:

Property. (The Basic Property of a Data Segment Hierarchy). Let DSH* be a data segment hierarchy for a database partition P with a set of udpate transactions T^u. Let t ϵ D_i in DSH*. Let d ϵ D_j be a data element in the access set of t (i.e., d ϵ w(t) U r(t).) Then either D_i = D_j or there exists a unique path between D_i and D_j in DSH*.

Proof. If $D_i \neq D_j$ then by the definition of data segment graph D_i -> D_j is contained in $DSG(P,T^u)$. Therefore by the definition of data segment hierarchy there must exist a path between D_i and D_j in DSH*. Since DSH* is a semi-tree, by definition, the path between D_i and D_j must be unique. *Q.E.D.*

For notational convenience, we also define a transaction classification which gives a name to a set of transactions rooted in the same data segment.

Definition. A *transaction classification* with respect to a data segment hierarchy DSH* for a database partition P and a set of update transactions T^u, is a partition of the set T^u into *transaction classes* T_1, T_2, ..., T_n, such that a transaction t ϵ T_i iff t is rooted in data segment D_i.

Therefore a transaction classification partitions the set of update transactions into classes, each of which corresponds to a data segment in the data partition and the notation '$t \in T_i$' has the same meaning as '$t \in D_i$'. Consequently, we will call accesses from a transaction to data elements in its own root data segment as *intra-class* accesses, while accesses from a transaction to data elements in data segments other than its own root data segment as *inter-class* accesses.

3.4 TYPES OF SYNCHRONIZATION PROTOCOLS

Based on the above definitions, a hierarchy of classes of update transactions is devised. A class rooted in a data segment may access a data segment that is higher than or lower than the root data segment or access data within its own root data segment. Since an update transaction is always rooted in the highest data segment that it writes into, an access to a higher data segment must be a read access, while the accesses to a lower or the root data segment could be either read or write. The rationale behind the HTS algorithm is the belief that it is possible to devise a synchronization protocol for accessing higher level data segments that is 'cheaper' than those accessing the root or the lower data segment. The purpose of such a hierarchy, therefore, is to make it possible to classify an update transaction's accesses to the database into accesses to a higher, lower or the root data segment, and

differentiate the synchronization protocols necessary for controlling each type of accesses.

In addition to the update transaction synchronization protocols, a protocol for synchronizing read-only transactions is also necessary. The goal is to enable a read-only transaction to access any data segment in the database without ever having to set read timestamps or be blocked or be aborted. Recall that the construction of a data segment hierarchy does not take into consideration the read-only transactions, therefore there is not a concept of 'rooting' that applies to the read-only transactions, and the protocols developed for the update transactions may not always apply.

In summary, we have identified four types of synchronization protocols each of which would apply to a type of database access:

(1) Synchronization protocol for reading a data segment higher than the root data segment of the transaction.

(2) Synchronization protocol for reading or writing the root data segment of the transaction.

(3) Synchronization protocol for reading or writing a data segment lower than the root data segment of the transaction.

(4) Synchronization protocol for reading any data segment by a read-only transaction.

3.5 NON-CYCLIC VS. CYCLIC DATABASE PARTITION

In this section, we will present a type of database partition that may result in a data segment hierarchy in which accesses to lower data segments by an update transaction do not exist, and therefore is the type of database partition that requires only two out of the three protocols for synchronizing update transactions listed in the previous section: the protocol for reading higher data segments and the one for accessing root data segment. As will be explained later, this would be the kind of database partitions that are most suited for applying the hierarchical timestamping algorithm.

Based on the topology of $DSG(P,T^u)$, a partition P can be of the cyclic type or the acyclic type as defined below.

Definition. A partition P of a database D into data segments is *acyclic (cyclic)* with respect to T^u if $DSG(P,T^u)$ is acyclic (cyclic).

We will give the properties of a non-cyclic database partition that explain why accesses to lower data segment from a transaction rooted in a higher data segment do not exist.

Properties of Non-cyclic Database Partitions

(1) Given a database partition P and its data segment graph $DSG(P,T^u)$ where $DSG(P,T^u)$ is non-cyclic, there exists a data segment hier-

archy DSH* such that the transitive closure of DSH* contains $DSG(P,T^U)$.

Proof. Since $DSG(P,T^U)$ is acyclic, it defines a partial order of the data segments. Any total order of the data segments that contains this partial order is a data segment hierarchy. Therefore there exists a data segment hierarchy DSH* such that the transitive closure of DSH* contains $DSG(P,T^U)$. *Q.E.D.*

We will refer to the DSH that satisfies the above condition for a non-cyclic database partition as a *natural* DSH for the partition. We will assume that for a non-cyclic partition the data segment hierarchy chosen would always be a natural DSH.

(2) Let p be an acyclic database parition with respect to T^U. Then $t \in T^U$ writes in one and only one data segment in P.

Proof. Suppose t writes in two distinct data segments D_i and D_j. Then according to our rule of construction of the data segment graph $DSG(P,T^U)$, $D_i \rightarrow D_j$, $D_j \rightarrow D_i \in DSG(P,T^U)$, therefore $DSG(P,T^U)$ cannot be acyclic which contradicts the assumption. *Q.E.D.*

(3) Let DSH* be a natural DSH for a non-cyclic database partition with update transaction set T^U. Then every access to a non-root data segment from any transaction in T^U is a read access to a higher data segment.

Proof. Based on the definition of a data segment hierarchy, every inter-class access to a higher data segment is a read access. Therefore we need only to show that no access to a low-

er data segment exists. This, however, is true because if it were not true, and suppose some transaction rooted in D_j needs to access some data element in D_i and D_i is lower than D_j in DSH*, then there exists $D_i \rightarrow D_j$ in $DSG(P,T^U)$ while $D_i \rightarrow D_j$ cannot be contained in the transitive closure of DSH*, which contradicts the definition of a natural data segment hierarchy. *Q.E.D.*

The above properties establish the fact that only two types of accesses exist for a non-cyclic database partition that has chosen a natural data segment hierarchy: those to the root data segment of the transaction or those (read accesses) to data segments that are higher than the root data segment of the transaction. In the following chapter the concurrency control protocols for these two types of accesses are developed and shown to be correct.

It is easy to see that for a cyclic database partition, the above properties no longer hold. Therefore, to apply the hierarchical timestamping algorithm to more generalized data base partitions, accesses from a transaction to lower data segments must be considered. In chapter five, we extend the theory provided in chapter four, and show that such accesses can be controlled by a third protocol, and without affecting the protocols developed in Chapter four, this set of three protocols for synchronizing update transactions collectively ensure the consistency of the database.

Finally, in chapter six, we develop the protocol for synchronizing read accesses from read-only transactions that are run in a database whose update transactions are synchronized by the above three protocols.

4.0 SYNCHRONIZING UPDATE TRANSACTIONS UNDER NON-CYCLIC PARTITIONS

Given a non-cyclical database partition, the two types of database accesses from an update transaction are the read and write accesses to its root data segment (intra-class accesses) and read acceses to higher data segments (inter-class accesses.) The key to our concurrency control technique is the recognition that, if a transaction t belongs to a class T_i that writes data segment D_i and reads data segment D_j, and D_j is *higher than* D_i in the Data Hierarchy Graph, then this transaction would appear to be a read only transaction so far as D_j is concerned. Therefore when a request to read a data element d in D_j is issued by t, there may exist a proper committed version of d that is safe to be given to t without the need of leaving a read timestamp with d. However, the way this proper version is computed must be such that the overall serializability is enforced. In other words, the introduction of transaction dependency of t on t', where t' is the transaction in class T_j which created the version of d that t is allowed to read, must never induce cycles in the transaction dependency graph as defined in Section 2.2. To this end, a function called the activity link function is devised to compute versions that inter-class read accesses may be granted, and a theorem which testifies to the correctness of this computation is presented. Based on this theorem, a concurrency control algorithm is also presented.

Notations.

(1) I(t) = the initiation time of a transaction t.

(2) C(t) = the commit time of a transaction t.

(3) TS(d^v) = the inititation time of the transaction that creates the version v of a data granule d, i.e., the write timestamp of d^v. (A data granule is the smallest unit that concerns the concurrency control component of the database system, and is the smallest unit of accesses so far as concurrency control is concerned.)

4.1 BASIC DEFINITIONS

The following definitions and properties assume a database with a non-cyclical partition with respect to T^u and has a natural database segment hierarchy DSH*.

Definition. A function I_i^{old} defined for a data segment D_i is a function which maps a time m to another time m', i.e., $m' = I_i^{old}(m)$, such that m' is the initiation time of the oldest active (i.e., uncommited and un-aborted) transaction rooted in the data segment D_i at time m. Formally,

$$I_i^{old}(m) = \begin{cases} m \text{ if there exists no } t \in D_i \text{ active at time } m, \\ \text{Min } (I(t)) \text{ otherwise, where} \\ \quad t \in T_i,\ I(t) < m \text{ and } C(t) > m. \end{cases}$$

Definition. Let the ***activity link function*** A_i^j be a function defined for a pair of data segments D_i and D_j, where $D_j \uparrow\!> D_i$ in DSH*. A_i^j recursively maps a time m to another time as follows.

$$A_i^j(m) = \begin{cases} I_j^{old}(m) \text{ if } D_i \rightarrow D_j = CP_i^j \\ A_k^j(A_i^k(m)) \text{ otherwise, where} \\ \quad D_i \rightarrow D_k \rightarrow \ldots \rightarrow D_j = CP_i^j. \end{cases}$$

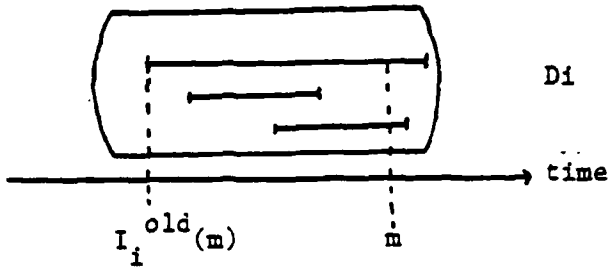
Intuitively, the purpose of function A_i^j is to map a time m corresponding to D_i to another time m' corresponding to D_j where versions in D_j before m' are considered 'safe' for a transaction rooted in D_i and started at time m to access. As an example, if the critical path between D_i and D_j is $D_i \rightarrow D_k \rightarrow D_j$, then $A_i^j(m) = I_j^{old}(I_k^{old}(m))$. *This is exemplified in Figure 8.*

4.2 CONCURRENCY CONTROL ALGORITHM FOR UPDATE TRANSACTIONS

Based on the definitions given above, we describe in this section the concurrency control algorithm for synchronizing update transactions under the hierarchical timestamping approach.

The algorithm is composed of two protocols, one for synchronizing intra-class accesses and one for synchronizing read-only accesses to higher data segments. The former is equivalent to the conventional

$I_i^{old}(m)$ = Initiation time of the oldest active transaction rooted in Di at time m:



$A_i^j(m) = I_j^{old}(I_k^{old}(m))$ if Di → Dk → Dj

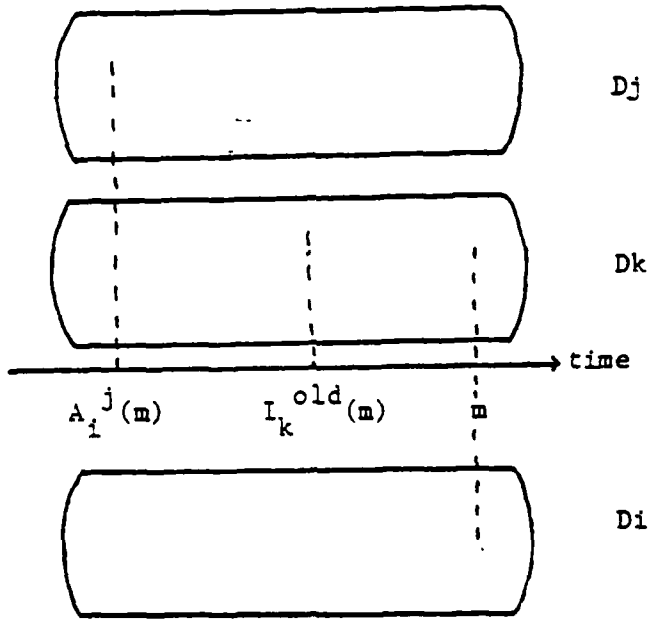


Figure 8. Graphical representation of the I-old function and the A function.

timestamp algorithm, where both read and write accesses will result in timestamping the accessed data element. The latter is a protocol which grants a particular version of the data element to the requesting transaction and involves no need for timestamping. This 'particular version' is chosen to be the latest version right before a certain time ceiling computed from the A function.

For the purpose of concurrency control, we assume that every data segment is controlled by a *segment controller* which supervises accesses to data granules within that segment.

Concurrency control algorithm for update transactions:

For every database access request from an update transaction $t \in T_i$ for a data granule $d \in D_j$, the following protocol is observed:

Protocol H

If $i \neq j$, then the segment controller of D_j provides the version d^o of d such that

$TS(d^o) = Max(TS(d^v))$ for all v such that

$TS(d^v) < A_i^j(I(t))$.

(Note that no trace of this access needs to be registered in any form for the purpose of concurrency control by the segment controller.)

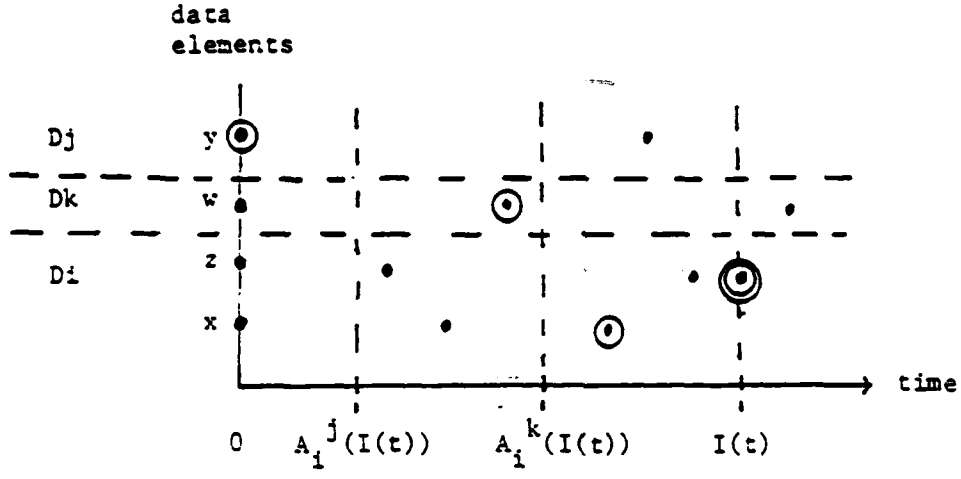
Protocol E

If i = j, then use the *basic timestamp ordering protocol* <Bernstein80> or the *multi-version timestamp ordering protocol* <Reed78>.

4.3 AN EXAMPLE

As an example, suppose a transaction t rooted in D_i needs to read data elements $x \in D_i$, $w \in D_k$, $y \in D_j$, and write data element $z \in D_i$, where $D_j \uparrow\!> D_k \uparrow\!> D_i$ in the data segment hierarchy. An execution of this transaction using protocols H and E is shown in Figure 9 where the dots represent versions of data elements and the circled dots represent versions to be accessed or created by t. In essence, to read x, t is given the version right before I(t) (the circled version of x in Figure 9) and I(t) is left as the read timestamp with this version of x. To read w, t is given the version right before $A_i^k(I(t))$ (the circled version of w in Figure 9) but no timestamp is left with this version. To read y, t is given the version right before $A_i^j(I(t))$ (again, the circled version of y in the figure) and no timestamp is left. Finally, to write z, the latest version of z is found and verified to be a version earlier than I(t) and which does not have a read timestamp greater than I(t). Then a new version of z is created with a timestamp I(t).

It can be seen that the difference between this execution and that of one using the conventional timestamp scheme lies with the way w and y



• : a version of a data element

⊙ : versions to be accessed by t

◎ : version to be created by t

t : a transaction rooted in Di with timestamp I(t) and reads x, w, y and writes z

Figure 9. Illustration of Applying Protocol H and Protocol E.

are accessed, which, under the HTS scheme, does not interfere at all with transactions that are concurrently updating w and y.

4.4 PROOF OF CORRECTNESS

To show that the above algorithm is correct, one must show that serializability is enforced. In order to do this, we follow the steps listed below:

(1) Define a relation called 'toplogically followa', denoted as '=>', between a pair of transactions and prove two properties of the relation.

(2) Show that these properties of the relation '=>' lead to Theorem 1, which states that if a concurrency control algorithm allows a transaction t_1 to directly depend on another transaction t_2 only if t_1 => t_2, then the transaction dependency graph is cycle free.

(3) Show that the two protocols introduced previously allows a t_1 to directly depend on a transaction t_2 only if t_1 => t_2, which concludes that the above algorithm preserves serializability.

Definition. A relation *topologically follows* (denoted as =>) is defined for a pair of transactions t_1, t_2, where $t_1 \in D_i$, $t_2 \in D_j$, D_i and D_j are connected by a critical path in a chosen data segment hierar-

chy DSH*, i and j not necessarily distinct. We say that t_1 ***topologically follows*** t_2 (or $t_1 \Rightarrow t_2$) iff

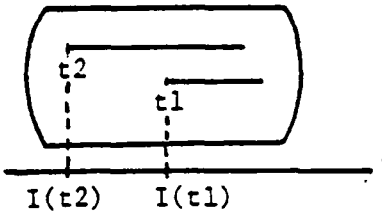
(1) if $D_i = D_j$ then $I(t_1) > I(t_2)$.

(2) If $D_i \uparrow\!> D_j$ then $I(t_1) \geq A_j^i(I(t_2))$.

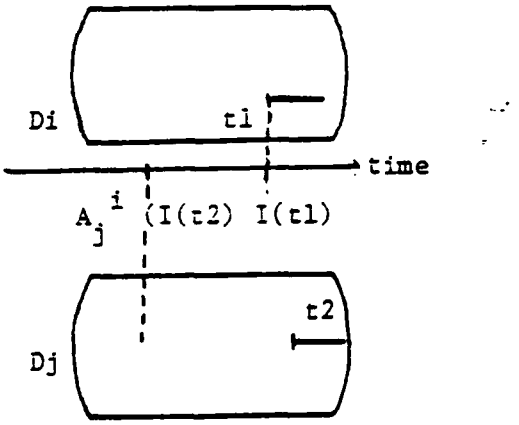
(3) If $D_j \uparrow\!> D_i$ then $I(t_2) < A_i^j(I(t_1))$.

Intuitively, => is a relation between transactions based on both the timing of the transactions and the hierarchical levels in the THG of the transaction classes that the transactions belong to. To be more specific, 't_1 => t_2' always means that t_1 is 'later' than t_2. However, this 'later' is not only based on the initiation times of the two transactions involved, but also on the relative levels of the data segments in which t_1 and t_2 are rooted: Given a fixed t_2, the lower the level of the data segment in which t_1 is rooted, the later t_1's initiation time has to be in order for $t_1 \Rightarrow t_2$ to hold. Clearly, => is defined only between transactions that are rooted in data segments that are on a critical path in DSH*, because otherwise the A function is undefined. This relation is exemplified in Figure 10. Two interesting properties concerning the relation => are presented below:

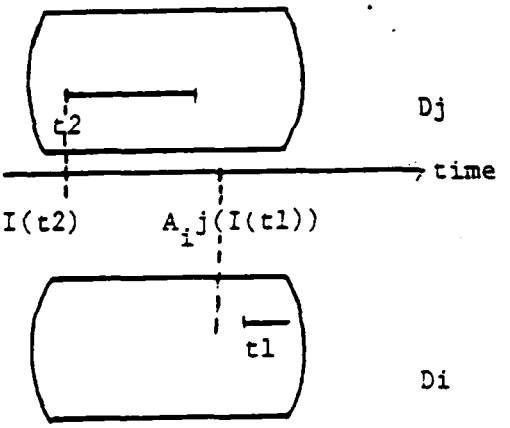
Property 1.1. The relation => is anti-symmetric. (This directly follows from the definition of the relation.)



(1) if Di = Dj
then I(t1) > I(t2)



(2) if Di higher than Dj
then $I(t1) \geq A_j i(I(t2))$



(3) if Dj higher than Di then
$I(t2) < A_i j(I(t1))$

Figure 10. Graphical representation of the relation t1 => t2.

Property 1.2 (The property of transitivity). The relation => is ***critical-path transitive***, i.e., if there exists $t_1 \in D_i$, $t_2 \in D_k$, $t_3 \in D_j$, such that $t_1 \Rightarrow t_2$, $t_2 \Rightarrow t_3$ and D_i, D_k and D_j are on a critical path in DSH*, then $t_1 \Rightarrow t_3$.

Proof. To prove this, we first give the following two obvious properties of the function A:

(0.1) If $D_j \not\leftrightarrow D_k \not\leftrightarrow D_i$ in DSH* then $A_k^j(A_i^k(m)) = A_i^j(m)$. (This directly follows from the fact that in DSH* of there exists one and only one critical path between any pair of data segments.)

(0.2) $A_i^j(m)$ has a non-negative slope (i.e., if $m > m'$ then $A_i^j(m) \geq A_i^j(m')$, and if $A_i^j(m) > A_i^j(m')$ then $m > m'$.)

We consider the following 5 groups of cases:

(1) $D_i = D_k = D_j$. By definition of => we have $I(t_1) > I(t_2) > I(t_3)$. Therefore $t_1 \Rightarrow t_3$.

(2) $D_i = D_k \neq D_j$. Two cases are considered:

(2.1) $D_i \not\leftrightarrow D_j$. Then $t_2 \Rightarrow t_3$ implies $I(t_2) \geq A_j^{-1}(I(t_3))$. $t_1 \Rightarrow t_2$ implies $I(t_1) > I(t_2)$. Therefore $I(t_1) > A_j^{-1}(I(t_3))$. Therefore $t_1 \Rightarrow t_3$.

(2.2) $D_j \not\leftrightarrow D_i$. Then $t_1 \Rightarrow t_2$ implies $I(t_1) > I(t_2)$. By Property 0.2 we have $A_i^j(I(t_1)) \geq A_i^j(I(t_2))$. $t_2 \Rightarrow t_3$ implies $A_i^j(I(t_2)) > I(t_3)$. Therefore $A_i^j(I(t_1)) > I(t_3)$. Therefore $t_1 \Rightarrow t_3$.

(3) $D_i \neq D_k = D_j$ Two cases are considered:

(3.1) $D_i \uparrow\!> D_j$. Then $t_2 \Rightarrow t_3$ implies $I(t_2) > I(t_3)$. By Property 0.2 we have $A_j^i(I(t_2)) \geq A_j^i(I(t_3))$. $t_1 \Rightarrow t_2$ implies $I(t_1) \geq A_j^i(I(t_2))$. Therefore $I(t_1) \geq A_j^i(I(t_3))$. Therefore $t_1 \Rightarrow t_3$.

(3.2) $D_j \uparrow\!> D_i$. Then $t_2 \Rightarrow t_3$ implies $I(t_2) > I(t_3)$. $t_1 \Rightarrow t_2$ implies $A_i^j(I(t_1)) > I(t_2)$. Therefore $A_i^j(I(t_1)) > I(t_3)$. Therefore $t_1 \Rightarrow t_3$.

(4) $D_i = D_j \neq D_k$. Two cases are considered:

(4.1) $D_i \uparrow\!> D_k$. Then $t_1 \Rightarrow t_2$ implies $I(t_1) > A_k^i(I(t_2))$. $t_2 \Rightarrow t_3$ implies $A_k^i(I(t_2)) > I(t_3)$. Therefore $I(t_1) > I(t_3)$. Therefore $t_1 \Rightarrow t_3$.

(4.2) $D_k \uparrow\!> D_i$. Then $t_1 \Rightarrow t_2$ implies $A_i^k(I(t_1)) > I(t_2)$. $t_2 \Rightarrow t_3$ implies $I(t_2) \geq A_i^k(I(t_3))$. Therefore $A_i^k(I(t_1)) > A_i^k(I(t_3))$. By Property 0.2 we have $I(t_1) > I(t_3)$. Therefore $t_1 \Rightarrow t_3$.

(5) $D_i \neq D_k \neq D_j$, $D_i \neq D_j$. Six cases are considered:

(5.1) $D_j \uparrow\!> D_k \uparrow\!> D_i$. Then $t_1 \Rightarrow t_2$ implies $A_i^k(I(t_1)) > I(t_2)$. From Property 0.1 and 0.2 we have $A_i^j(I(t_1)) = A_k^j(A_i^k(I(t_1))) \geq A_k^j(I(t_2))$. Therefore $A_i^j(I(t_1)) \geq A_k^j(I(t_2))$. $t_2 \Rightarrow t_3$ implies $A_k^j(I(t_2)) > I(t_3)$. Therefore $A_i^j(I(t_1)) > I(t_3)$. Therefore $t_1 \Rightarrow t_3$.

(5.2) $D_i \uparrow\!> D_k \uparrow\!> D_j$. Then $t_2 \Rightarrow t_3$ implies $I(t_2) \geq A_j^k(I(t_3))$. From Property 0.1 and 0.2 we have $A_k^i(I(t_2)) \geq A_k^i(A_j^k(I(t_3))) = A_j^i(I(t_3))$. $t_1 \Rightarrow t_2$ implies $I(t_1)) \geq A_k^i(I(t_2))$. Therefore $I(t_1) \geq A_j^i(I(t_3))$. Therefore $t_1 \Rightarrow t_3$.

(5.3) $D_j \uparrow\!> D_i \uparrow\!> D_k$. Then $t_1 \Rightarrow t_2$ implies $I(t_1) \geq A_k^i(I(t_2))$. From Property 0.1 and 0.2 we have $A_i^j(I(t_1)) \geq A_i^j(A_k^i(I(t_2)))) = A_k^j(I(t_2))$. $t_2 \Rightarrow t_3$ implies $A_k^j(I(t_2)) > I(t_3)$. Therefore $A_i^j(I(t_1)) > I(t_3)$. Therefore $t_1 \Rightarrow t_3$.

(5.4) $D_k \uparrow\!> D_i \uparrow\!> D_j$. Then $t_1 \Rightarrow t_2$ implies $A_i^k(I(t_1)) > I(t_2)$. $t_2 \Rightarrow t_3$ implies $I(t_2) \geq A_j^k(I(t_3))$. From Property 0.1 and 0.2 we have $A_i^k(I(t_1)) > A_j^k(I(t_3)) = A_i^k(A_j^i(I(t_3)))$ Therefore $I(t_1) > A_j^i(I(t_3))$. Therefore $t_1 \Rightarrow t_3$.

(5.5) $D_k \uparrow\!> D_j \uparrow\!> D_i$. Then $t_1 \Rightarrow t_2$ implies $A_i^k(I(t_1)) > I(t_2)$. $t_2 \Rightarrow t_3$ implies $I(t_2) \geq A_j^k(I(t_3))$. Therefore $A_i^k(I(t_1)) > A_j^k(I(t_3))$. However, $A_i^k(I(t_1)) = A_j^k(A_i^j(I(t_1)))$. Therefore $A_i^j(I(t_1)) > I(t_3)$. Therefore $t_1 \Rightarrow t_3$.

(5.6) $D_i \uparrow\!> D_j \uparrow\!> D_k$. Then $t_1 \Rightarrow t_2$ implies $I(t_1) \geq A_k^i(I(t_2))$. But $A_k^i(I(t_2)) = A_j^i(A_k^j(I(t_2)))$. And $t_2 \Rightarrow t_3$ implies $A_k^j(I(t_2)) > I(t_3)$. Therefore $A_k^i(I(t_2)) \geq A_j^i(I(t_3))$. Therefore $I(t_1) \geq A_j^i(I(t_3))$. Therefore $t_1 \Rightarrow t_3$.

In each group, we have permutated the order of levels among the distinct transaction classes to arrive at a total 13 cases. These cases exhaust all the possible situations that govern t_1, t_2 and t_3 and for every situation transitivity is shown to hold. Therefore we conclude that $\Rightarrow$ is critical-path transitive. *Q.E.D.*

We now define the following synchronization rule and show that if a concurrency control algorithm enforces this rule then the transaction dependency graph is cycle free.

Definition. We say that the ***partition synchronization rule*** (abbreviated as PSR) is enforced in a schedule S of a set of transactions T given a data segment hierarchy DSH* if, for any t_1, $t_2 \in T$, $t_1 \rightarrow t_2 \in TG(S)$ implies that $t_1 \Rightarrow t_2$.

In other words, a concurrency control algorithm enforces the partition synchronization rule if it allows direct dependencies to occur between transactions t_1 and t_2 only if $t_1 \Rightarrow t_2$.

We now prove that a concurrency control algorithm that enforces PSR is also correct. This involves proving that a concurrency control algorithm that enforces PSR will only produce schedules whose transaction dependency graph is cycle-free. The following theorem therefore constitutes our proof.

Theorem 1. Let TG(S) be a transaction dependency graph of a schedule S of a set of udpate transations T^u run on a database with a non-cyclic partition P, and the schedule S observes the partition synchronization rule with respect to a natural data segment hierarchy DSH*, then TG(S) has no cycles.

Proof. In order to prove Theorem 1, we first give the following two definitions and a lemma about the transaction dependency graph.

Definition. A *critical path dependency*, between two distinct transactions $t_1 \in D_i$ and $t_2 \in D_j$, denoted as $CD(t_1, t_2)$, is a cycle-free dependency path from t_1 to t_2 in TG(S) and D_i and D_j are on a critical path in DSH*, i and j not necessarily distinct.

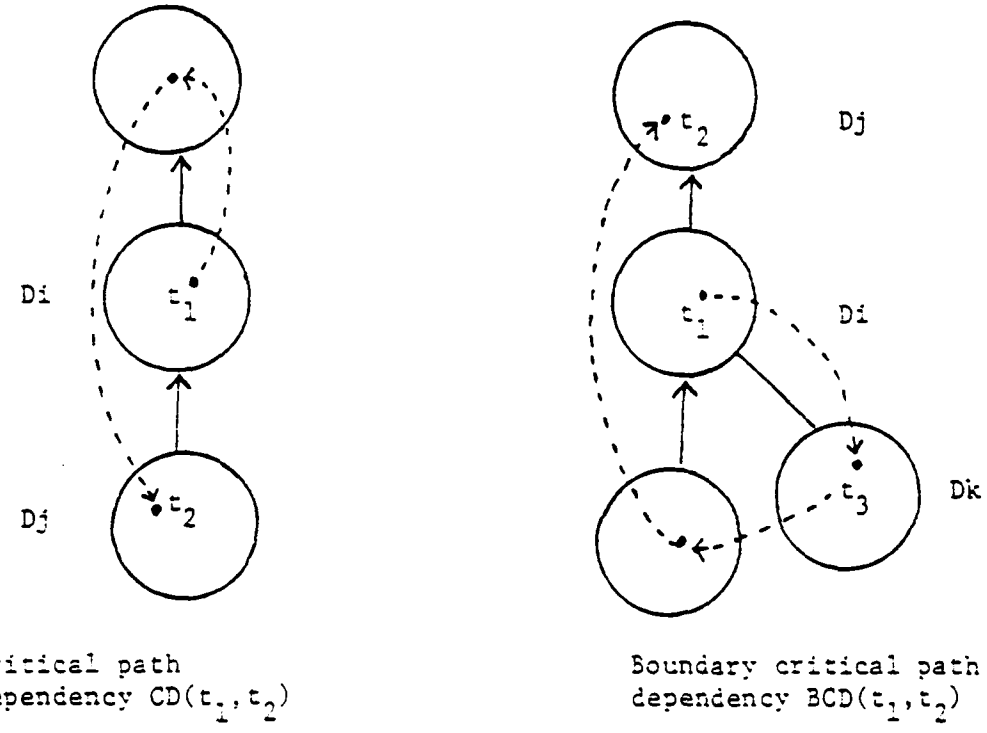
Definition. A *boundary critical path dependency* in TG(S) between two transactions $t_1 \in D_i$ and $t_2 \in D_j$, where $t_1 \neq t_2$, denoted as $BCD(t_1, t_2)$, is a $CD(t_1, t_2)$ such that either or both of the following are true:

(1) There exists $t_3 \in T_k$ such that $t_1 \rightarrow t_3 \in CD(t_1,t_2)$ and D_i, D_j and D_k are *not* on one critical path;

(2) There exists $t_4 \in D_l$ such that $t_4 \rightarrow t_2 \in CD(t_1,t_2)$ and D_i, D_j and D_l are *not* on one critical path.

These two concepts are illustrated in Figure 11.

Property. If $BCD(t_1, t_4)$, where $t_1 \in D_i$ and $t_4 \in D_j$, then there exist $t_2 \in D_k$ and $t_3 \in D_l$, t_2, t_3 not necessarily distinct, such that $CD(t_1, t_2) \subset CD(t_1, t_4)$, $CD(t_2,t_3) \subset CD(t_1, t_4)$, $CD(t_3,t_4) \subset CD(t_1, t_4)$ and D_i, D_j, D_k and D_l are on one critical path in DSH*. (This directly follows from the fact that DSH* is a semi-tree.)



Critical path dependency $CD(t_1, t_2)$

Boundary critical path dependency $BCD(t_1, t_2)$

• : a transaction

.....> : direct transaction dependency

Figure 11. Illustration of Critical Path Dependency (CD) and Boundary CD (BCD).

Lemma 1. If there exists a critical path dependency $CD(t_1,t_2)$ in a transaction dependency graph TG(S(T)) where the schedule S enforces the partition synchronization rule, then $t_1 \Rightarrow t_2$.

Proof. Let ℓ be the length (in number of arcs, i.e., direct dependencies) of a critical path dependency. Then ℓ has a total order and is bounded from below by 1. By way of complete mathematical induction, to prove that if $CD(t_1,t_2)$ then $t_1 \Rightarrow t_2$, we have to show the following:

(1) If $\ell(CD(t_1,t_2)) = 1$ then $t_1 \Rightarrow t_2$.

(2) If $\ell(CD(t_1,t_2)) = g$ and if $t_a \Rightarrow t_b$ for all t_a, t_b s.t. there exists $CD(t_a,t_b)$ and $\ell(CD(t_a,t_b)) < g$, then $t_1 \Rightarrow t_2$.

Now we prove the above two statements.

(1) In this case, $CD(t_1,t_2) = t_1 \rightarrow t_2$. Since S enforces the partition synchronization rule, by definition, we have $t_1 \Rightarrow t_2$.

(2) To prove the second statement, let $t_3 \in D_k$ and $t_4 \in D_l$ be such that $t_1 \rightarrow t_3 \in CD(t_1, t_2)$, $t_4 \rightarrow t_2 \in CD(t_1, t_2)$, and a path, denoted as $Path(t_3,t_4)$, from t_3 to t_4 such that $Path(t_3,t_4) \subset CD(t_1,t_2)$. Also let $t_1 \in D_i$ and $t_2 \in D_j$. Consider the following two cases:

(2.1) If $CD(t_1,t_2)$ is *not* a BCD, then $Path(t_3,t_4)$ is a $CD(t_3,t_4)$. Since $\ell(CD(t_1,t_2)) < g$ therefore $t_1 \Rightarrow t_2$. And by the definition of CD, D_i, D_j, D_k and D_l must be on one critical path of DSH*. Therefore we have $t_1 \rightarrow t_3$, $t_4 \rightarrow t_2$ and $t_3 \Rightarrow t_4$. By Property 1.2 (i.e., the property of critical path transitivity) we have $t_1 \Rightarrow t_2$.

(2.2) If $CD(t_1,t_2)$ is a BCD, then by the property above of a BCD we have that there exist $t_5 \in D_m$ and $t_6 \in D_n$ such that $CD(t_1,t_5) \subset CD(t_1,t_2)$, $CD(t_5,t_6) \subset CD(t_1,t_2)$, and $CD(t_6,t_2) \subset CD(t_1,t_2)$, where D_m, D_n, D_i and D_j are on one critical path of DSH*. Since $\ell(CD(t_1, t_5)) < g$, therefore $t_1 \Rightarrow t_5$. Similarly, $t_6 \Rightarrow t_2$ and $t_5 \Rightarrow t_6$. By Property 1.2 we conclude $t_1 \Rightarrow t_2$. *Q.E.D.*

Now we are ready to prove Theorem 1.

Theorem 1.

Proof. Suppose there exists a cycle. Then the cycle involves at least two transactions t_1 and t_2 that belong to transactions that are on one critical path. This means that there exist $CD(t_1,t_2)$ and $CD(t_2,t_1)$. By the above lemma, $CD(t_1,t_2)$ implies $t_1 \Rightarrow t_2$ and $CD(t_2,t_1)$ implies $t_2 \Rightarrow t_1$. However, => is anti-symmetric (by &propii.). Therefore $t_1 \Rightarrow t_2$ and $t_2 \Rightarrow t_1$ cannot be true at the same time. Therefore there can be no cycle in this transaction dependency graph. *Q.E.D.*

We conclude that if a concurrency control algorithm enforces the Partition Synchronization Rule then it is correct. What is left is to show is that Protocols H and E enforce this rule. Therefore we must show that, by employing these protocols, $t_1 \rightarrow t_2$ implies $t_1 \Rightarrow t_1$. This is translated into the following three cases:

(1) If t_1 and t_2 are rooted in the same data segment, the algorithm must allow t_1 to read a version v of a data granule d created by t_2, or to create a new version of a data granule d whose latest version d^v was created by t_2, only if t_2 has an inititation time that is less than that of t_1. (i.e., only if $TS(d^v) < I(t_1)$.)

Protocol B of our algorithm satisfies this requirement.

(2) If t_1 is rooted in D_i of a lower level while t_2 in D_j of a higher level, then the algorithm must allow t_1 to read d^v created by t_2 only if t_2 has an initiation time less than $A_i^j(I(t_1))$. (i.e., only if $TS(d^v) < A_i^j(I(t_1))$.)

Protocol A of our algorithm satisfies this requirement.

(3) If t_1 is rooted in D_i of a higher level while t_2 in D_j of a lower level, then the algorithm must allow t_1 to create, at time m, a new version of a data granule whose predecessor d^v has been read by t_2, only if t_1 has an initiation time greater than or equal to $A_j^i(I(t_2))$.

This, however, is always true because, by the very fact that t_2 by time m has already read d^v we know that $I(t_2) < m$, and therefore $A_j^i(I(t_2)) \leq A_j^i(m)$. Also, because of the fact that t_1 is active at m (i.e., not committed yet at m) we know that $A_j^i(m) < I(t_1)$. This leads to the conclusion that $I(t_1) \geq A_j^i(I(t_2))$.

Therefore we conclude that our algorithm enforces PSR, which completes our proof.

5.0 SYNCHRONIZING UPDATE TRANSACTIONS UNDER CYCLIC PARTITION

In this chapter we will extend the algorithm developed in the previous chapter, which handles the special case of a non-cyclic parition, to include the capability of handling cyclic partitions. As mentioned before, the difference between a non-cyclic and a cyclic partition is that in the former the only inter-class accesses are the read-only accesses to higher data segments, while in the latter the inter-class accesses include reads and writes to lower data segments as well. In the theorems proven in this chapter, we will show that the existence of accesses to lower data segments in a data segment hierarchy can be treated independently from the treatment of those that are intra-class or to higher data segments. That is, to deal with non-cyclic partitions in which accesses to lower data segments exist, one needs only to add another protocol on top of the protocols already devised in the previous chapter for intra-class and higher-level accesses.

We will first give the definitions of some functions that are needed in describing the protocol. Then we provide a description of the protocol, followed by an example of the use of the protocols and the proof of correctness.

5.1 BASIC DEFINITIONS

The following definitions assume that a data segment hierarchy DSH* is given for a cyclic database partition. To compute the timestamps a transaction uses to access data segments lower than the transaction's own root segment, we will now describe the functions C_i^{late} and B_j^i that can be considered conceptually the *inverse* of functions I_i^{old} and A_i^j. These functions are illustrated in Figure 12.

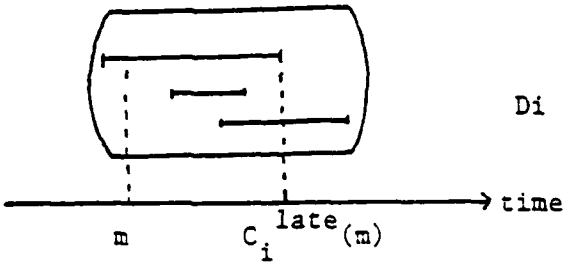
Definition. Let C_i^{late}: $m \rightarrow m'$ be a function which maps a time m to another m' where D_i is a data segment and $C_i^{late}(m)$ is determined as follows.

$$C_i^{late}(m) = \begin{cases} m \text{ if there exists no } t \in D_i \text{ active at time } m, \\ \text{Max } (C(t)) \text{ otherwise, where} \\ \quad t \in D_i,\ I(t) < m \text{ and } C(t) > m. \end{cases}$$

That is, $C_i^{late}(m)$ is the *latest* commit time of all transactions rooted in D_i that started before time m.

While the A function maps a time in a lower segment to the initiation time of some transaction rooted in a higher segment, the B function maps a time in a higher segment to the commit time of some transaction rooted in a lower segment:

$C_i^{late}(m)$ = Commit time of the transaction that commits the latest among all transactions rooted in Di active at time m



$B_i^j(m) = C_k^{late}(C_i^{late}(m))$ if Dj → Dk → Di

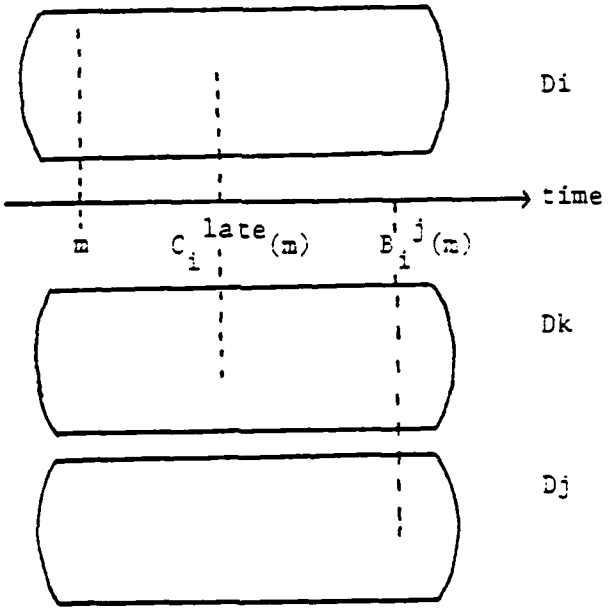


Figure 12. Illustration of Function C-late and Function B.

Definition. The *Backward activity link function*, defined for a pair of data segments D_i and D_j, where $D_j \uparrow\!> D_i$, denoted as $B_j^i(m)$, is a function which maps a time value m to another such that

$$B_j^i(m) = \begin{cases} m \text{ if } D_i = D_j \\ C_j^{late}(m) \text{ if } D_i \rightarrow D_j = CP_i^j \\ B_k^i(B_j^k(m)) \text{ otherwise, where } D_i \rightarrow \ldots \rightarrow D_k \rightarrow D_j = CP_i^j. \end{cases}$$

5.2 THE SYNCHRONIZATION PROTOCOL

Now we introduce the hierarchical timestamp protocol for cyclic partition.

Hierarchical Timestamp Protocol For Update Transactions:

For every database access request from an update transaction $t \in D_i$ for a data granule $d \in D_j$, the following protocol is observed:

Protocol E (for accessing a segment Equal to root segment)

If $D_j = D_i$, then

(1) If it is a read request, then grant the latest version before I(t) of d, and leave I(t) as the read timestamp of this version of d if its current read timestamp is smaller than I(t).

(2) If it is a write request, then if the read timestamp of the latest version before I(t) of d is smaller than I(t), then create a new version of d with version number I(t). Otherwise abort t.

Protocol H (For accesssing Higher segments)

If D_j $\uparrow>$ D_i then grant t access to the latest version before $A_i^j(I(t))$ of d.

Protocol L (For accessing Lower segments)

If D_i $\uparrow>$ D_j, then

(1) If it is a read request, then grant the latest version before $B_i^j(I(t))$ of d, and leave $B_i^j(I(t))$ as the read timestamp of this version of d if its current read timestamp is smaller than $B_i^j(I(t))$.

(2) If it is a write request, then if the read timestamp of the latest version before $B_i^j((t))$ of d is smaller than $B_i^j(I(t))$, then create a new version of d with version number $B_i^j(I(t))$. Otherwise abort t.

Several observations are made of this set of protocols:

(i) If the database segment hierarchy DSH* consists of a single data segment, then only Protocol E will apply, and the hierarchical timestamp algorithm is reduced to the conventional MVTS algorithm.

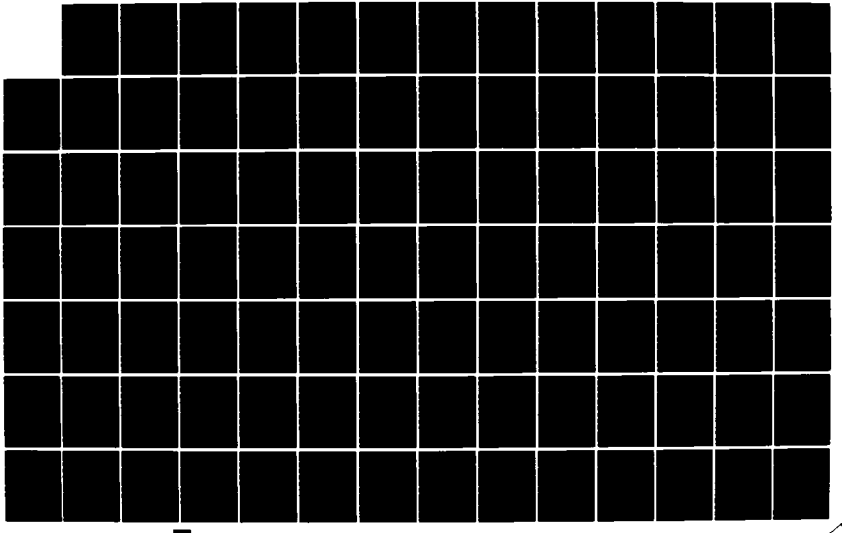
(2) Since no transaction will write a data segment higher than its own root segment, Protocol H needs to cover only read accesses. More importantly, Protocol H is 'cheaper' than either Protocol E or Protocol L, since it does not require timestamping the data element accessed.

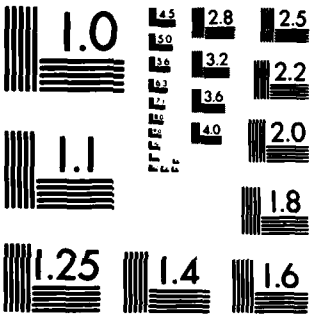
(3) Protocol L is essentially the same as Protocol E with the exception that the timestamps used are different from the timestamps of the accessing transactions. Since $B_i^j(I(t)) \geq I(t)$, Protocol L is the most 'expensive' among the three, in addition to the difficulty discussed in the following paragraph.

There is a difficulty associated with implementing Protocol L. To compute $B_i^j(I(t))$ as a timestamp to synchronize access requests from a transaction t is a tricky matter since theoretically $B_i^j(I(t))$ is not 'computable' until at least after transaction t has committed. This dilemma can be resolved by artificially computing (i.e., 'guessing') the value of B functions and enforcing it at a later time. To do so, an algorithm is designed to compute pseudo-C_i^{late} values, and another algorithm, which is a recursive application of the first algorithm, is used to computer pseudo-B_i^j values. The first algorithm returns the psuedo-C_i^{late} value by adding a constant $a_i > 0$, generic to a data segment D_i, to the argument value. In addition, it inserts constraints for concurrency control mechanism to enforce the validity of this computation at a later time. These two algorithms are as follows:

AD-A150 612 THE HIERARCHICAL DATABASE DECOMPOSITION APPROACH TO DATABASE CONCURRENCY..(U) ALFRED P SLOAN SCHOOL OF MANAGEMENT CAMBRIDGE MA CENTER FOR I.. M HSU DEC 84 2/4

UNCLASSIFIED CISR-TR-16 N00039-83-C-0463 F/G 9/2 NL





pseudo_C (D_i, m): Function.

Insert a pseudo transaction t' ϵ D_i such that I(t') = m and C(t') = m + a_i.

Insert constraint: Abort all t" ϵ D_i s.t. I(t") < m and C(t") > m + a_i.

Return (m + a_i).

(Assume $D_i \rightarrow D_{i1} \rightarrow D_{i2} \rightarrow \ldots \rightarrow D_j = CP_i^j$)

pseudo_b (D_i, D_j, m): Function

B = pseudo_c (D_{i1}, pseudo_c (D_{i2}, ... pseudo_c (D_j, m) ...)).

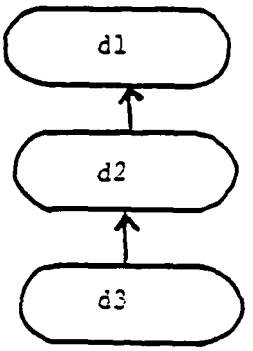
Return (B);

If a_i's are selected in such a way that a large portion of the non-pseudo transactions rooted in D_i can be executed in time less than a_i, the chance of having to abort transactions in D_i in order to enforce the constraint (i.e., step 2 of the algorithm pseudo_c) is relatively small.

5.3 AN EXAMPLE

In this section we illustrate the use of protocols H, E and L for controlling update transactions through a simple banking example. The scenario is as follows. The database is composed of three types of information about customers: overdraft limits, demand deposit account

Data Segment Hierarchy:



Overdraft Limits (OL)
D_1

Account Balance (AB)
D_2

Loan Balance (LB)
D_3

(d1 = Smith's OL, d2 = Smith's AB, d3 = Smith's LB)

Four transactions:

t1: Increment Smith's Overdraft limit (Read d1, write d1)
(t1 rooted in D_1)

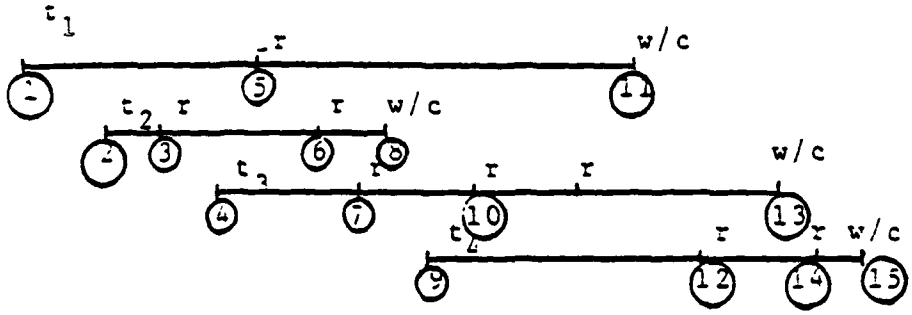
t2: Withdraw from Smith's Account (Read d1, d2, write d2)
(t2 rooted in D_2)

t3: Approve laon for Smith subject to current account status
(Read d2, d1, d3, write d3) (t3 rooted in D_3)

t4: Decrement Smith's overdraft limit (Read d1, d2, write d1)
(t4 rooted in D_1)

Figure 13. Scenario of an example illustrating the HDD update protocols.

Timing of events:



Execution and control sequence:

Protocol used	Event No.	Event and Control
H	3	t2 reads d1, granted version 0, since it is before $A_2^1(I(t2))$
E	5	t1 reads d1, granted version 0 and leaves RTS = I(t1)
E	6	t2 reads d2, granted version 0 and leaves RTS = I(t2)
H	7	t3 reads d2, granted version 0, since it is before $A_3^2(I(t3))$
E	8	t2 creates a new version of d2 (version 1) and commits
H	10	t3 reads d1, granted verison 0, since it is before $A_3^1(I(t3))$
E	11	t1 creates new version of d1 (version 1) and commits
E	12	t4 reads d1, granted version 1, and leaves RTS = I(t4)
E	13	t3 creats new version of d3 (version 1) and commits
E	14	t4 reads d2, granted version 1 and leaves RTS = I(t4) + c
E	15	t4 creats new version of d1 (version 2) and commits

Figure 14. Timing and control responses of the example illustrating the HDD update protocols.

balances (or simply 'account balances'), and loan account balances. These types of information are organized into three data segments with a data segment hierarchy shown in Figure 13. Also, three types of update transactions are run against the database: overdraft limit update, deposit or withdrawal against a demand deposit account, and loan payment or new loan approvals. In the interleaved transaction execution to be demonstrated here, a set of four transactions are run concurrently, and the tasks involved in these transactions are also shown in Figure 13, where d1, d2 and d3 are, respectively, the overdraft limit, the account balance and the loan balance of a customer Mr. Smith's. The timing of the interleaved read and write requests of these transactions to be input to the concurrency control algorithm are shown in Figure 14. Assuming that the current versions of d1, d2 and d3 are all version 0 and are all before the initiation time of the earliest transaction in the set, i.e., I(ti), the control responses that the concurrency control facility would generate for each request are shown in Figure 14. As shown in the figure, every request is granted at the time the request is issued and, in this example, no blocking nor aborts are induced.

It is interesting to note that the same interleaved schedule of read/write requests, if controlled by the two-phase locking algorithm, would result in blocking and a deadlock. Specifically, when t3 issued its 'read d1' request at time 10, d1 was at that time read-locked by t1. Since read requests do not conflict, t3 would be granted access to d1 and an additional read lock, owned by t3, would be imposed on d1. How-

ever, when t1 issued the 'write d1' request at time i1, it would be forced to wait since it could not convert its read lock on d1 to a write lock when d1 was at that time also read-locked by t3. Therefore, t1 could not commit and must wait for t3 to free its read lock on d1. However, when t3 finally committed and freed its read lock on d1 at time i3, d1 had in the mean time also been read-locked by t4 at time i2, therefore t1 would now be forced to wait for t4 to commit. Unfortunately, t4 could not commit at time i5 because it also needed to write d1 and d1 was held read-locked by t1. Therefore, t1 and t4 deadlocked at time 15, and one of the two transactions must be backed out and restarted.

Similarly, if this interleaved schedule of read/write requests is given to a conventional MVTS aglorithm, some aborts of transactions would result. In particular, t1 would be aborted at time i1 in attempting to write d1 and t2 would be aborted at time 8 in attempting to write d2. Therefore this example also demonstrates a scenario in which the the HDD protocols would have performed better in terms of level of concurrency than the two-phase locking algorithm and the conventional multi-version timestamping algorithm.

5.4 PROOF OF CORRECTNESS

Given the rules for testing for serializability (i.e., to show that the transaction dependency graph as defined before is non-cyclic if the above protocol is enforced,) the following steps are devised to prove correctness of the HTS algorithm:

(1) Show that direct dependencies may occur only between transactions that are rooted in segments that are related in DSH*.

(2) Making use of Theorem 1 proven in the previous chapter which asserts that if a schedule S enforces the relation 'topologically follows', i.e., the relation '=>' as defined in the proof of the previous chapter, on all direct transaction dependencies (i.e., $t_2 \rightarrow t_1 \in TG(S)$ only if $t_2 \Rightarrow t_1$) then the transaction dependency graph TG(S) has no cycle, show that the HTS algorithm produces only schedules that enforce the relation '=>' on all direct transaction dependencies.

Lemma 2. Given a DSH* and let $t_1 \in D_i$ and $t_2 \in D_j$. If $t_2 \rightarrow t_1$, then D_i and D_j are related in DSH*.

Proof. Let $d \in D_k$ be the contended data element that causes $t_2 \rightarrow t_1$. Since at least one transaction writes d, we have either $D_i \rightarrow(=) D_k \rightarrow(=) D_j$ contained in $DSG(P,T^u)$ or $D_j \rightarrow(=) D_k \rightarrow(=) D_i$ contained in $DSG(P,T^u)$. By the definition of data segment hierarchy, D_i and D_j must be related in DSH*.

Given the Theorem 1 which states that the relation => can be used as a vehicle for ordering transactions for concurrency control purposes, the following theorem completes our proof of correctness.

Theorem 2. Let S be a schedule that is permitted by the hierarchical timestamp protocol. Then $t_2 \rightarrow t_1 \in TG(S)$ only if $t_2 \Rightarrow t_1$. (i.e., The hierarchical timestamp protocol enforces PSR under a cyclic partition.)

Proof. We will first present the following 3 properties which bind the functions A and B together are used to transform the timing relationship imposed by the protocol to that defined by the relation =>:

Property 2.1. $A_i^j(B_j^i(m)) \geq m$, where $D_i \rightarrow D_{i1} \rightarrow \rightarrow \ldots \rightarrow D_{i(n-1)} \rightarrow D_{in} \rightarrow D_j = CP_i^j$ in the data segment hierarchy DSH*.

Proof. $A_i^j(B_j^i(m)) = A_i^j(C_{i1}(\ldots(C_{in}(C_j(m)))\ldots))$. ($C_j$ is an abbreviated expression for C_j^{late} and I_j is an abbreviated expression for I_j^{old}) Let $m_j = C_j(m)$. Then $m_j = C(t_j^o)$ if there exists $t \in D_j$ active at time m and $C_j(m) = C(t_j^o)$, and $m_j = m$ if there exists no $t \in D_j$ active at time m. Therefore $A_i^j(B_j^i(m)) = A_i^j(C_{i1}(\ldots(C_{in}(m_j))\ldots))$. Continue substitution of the L function in the expression with similarly defined $m_{in}, \ldots, m_{i1}$, we get $A_i^j(B_j^i(m)) = A_i^j(m_{i1})$. Now we start spelling out the function A_i^j: $A_i^j(m_{i1}) = A_{i1}^j(I_{i1}(m_{i1}))$. Consider the following two cases:

(1) If there exists no $t \in D_{i1}$ active at m_{i1}, then $I_{i1}(m_{i1}) = m_{i1}$. Since $m_{i1} = C_{i1}(m_{i2}) \geq m_{i2}$, we have $I_{i1}(m_{i1}) \geq m_{i2}$.

(2) If there exists $t \in D_{i1}$ active at m_{i1}, then $I_{i1}(m_{i1}) = I(t_{i1}')$, where $t_{i1}' \neq t_{i1}^{o}$ (since t_{i1}' is active at m_{i1} while t_{i1}^{o} commits at m_{i1}), and $I(t_{i1}') \geq m_{i2}$ (since if $I(t_{i1}') < m_{i2}$ then during the previous application of C_{i1}, $C_{i1}(m_{i2})$ should be equal to $C(t_{i1}')$ and not $C(t_{i1}^{o})$, and contradicts the assumption.) Therefore $I_{i1}(m_{i1}) \geq m_{i2}$.

Therefor we conclude $I_{i1}(m_{i1}) \geq m_{i2}$. By the same reasoning we continue spelling out the A function to arrive at the following: $A_i^j(B_j^i(m)) = A_{in}^j(I_{in}(m_{in}))$. Since $I_{in}(m_{in}) \geq m_j = C_j(m)$, we have $A_i^j(B_j^i(m)) \geq A_{in}^j(C_j(m))$. Since $A_{in}^j(C_j(m)) = I_j(C_j(m)) \geq m$, we have $A_i^j(B_j^i(m)) \geq m$. *Q.E.D.*

Property 2.2. $A_i^j(B_j^i(m) - \epsilon) < m$, where $D_i \rightarrow D_{i1} \rightarrow \rightarrow \ldots \rightarrow D_{i(n-1)} \rightarrow D_{in} \rightarrow D_j = CP_i^j$ in the data segment hierarchy DSH*, and ϵ a small value.

Proof. Let m_j, m_{in}, ..., m_i be defined in the same way as in the proof of Property 2.1. We have $A_i^j(B_j^i(m)-\epsilon) = A_i^j(m_{i1} - \epsilon) = A_{i1}^j(I_{i1}(m_{i1} - \epsilon))$. Now we show that $I_{i1}(m_{i1} - \epsilon) < m_{i2}$. Consider the following two cases:

(1) If there exists no $t \in D_{i1}$ active at m_{i2}, then $m_{i1} = C_{i1}(m_{i2}) = m_{i2}$. Therefore $I_{i1}(m_{i1} - \epsilon) = I_{i1}(m_{i2} - \epsilon) \leq m_{i2} - \epsilon < m_{i2}$.

(2) If there exists $t \in D_{i1}$ active at m_{i2}, then $m_{i1} = C_{i1}(m_{i2}) = C(t_{i1}^{o})$ where $I(t_{i1}^{o}) < m_{i2}$. Therefore $I_{i1}(m_{i1} - \epsilon) = I_{i1}(C(t_{i1}^{o}) - \epsilon) \leq I(t_{i1}^{o}) < m_{i2}$.

Therefore we conclude $I_{i1}(m_{i1} - \epsilon) < m_{i2}$. Let $m_{i1}' = I_{i1}(m_{i1} - \epsilon)$. Then $m_{i1}' < m_{i2}$, and $A_i^j(B_j^i(m) - \epsilon) = A_{i1}^j(m_{i1}')$. Continue the process of substitution we have $A_i^j(B_j^i(m) - \epsilon) = A_{in}^j(m_{in}') = I_j(m_{in}')$ where $m_{in}' < m_j$. But $I_j(m_{in}') \leq I_j(m_j - \epsilon) = I_j(C_j(m) - \epsilon) < m$. Therefore $A_i^j(B_j^i(m) - \epsilon) < m$. *Q.E.D.*

Property 2.3. $A_i^j(B_j^i(I(t))) > I(t)$ where $D_j \uparrow\!> D_i$ and $t \in D_i$ is a transaction.

Proof. Consider the following two cases:

(1) If $D_i \rightarrow D_j$ is a critical arc in DSH*, then $A_i^j(B_j^i(I(t))) = I_j(C_j(I(t)))$. Since all transactions that start at or before $I(t)$ have committed by the time indicated by $C_j(I(t))$, one concludes that $I_j(C_j(I(t))) > I(t)$.

(2) If $D_i \rightarrow \ldots \rightarrow D_{in} \rightarrow D_j \subset$ DSH*, then $A_i^j(B_j^i(I(t))) = I_j(A_i^{in}(B_{in}^i(C_j(I(t)))))$. By Property 2.1, we have the above expresssion $\geq I_j(C_j(I(t)))$, which by (1) is greater than $I(t)$. *Q.E.D.*

Now we are ready to prove Thereom 2. Let $t_2 \in D_j$ and $t_1 \in D_i$, and $t_2 \rightarrow t_1$ due to a contended data element $d \in D_k$. To show that $t_2 \rightarrow t_1$ implies $t_2 \Rightarrow t_1$, we must consider all the possible permutations in the hierarchical relationship in DSH* among D_i, D_j, and D_k. Since at least one of the transactions t_1 and t_2 must have write accesses to D_k, therefore the permutations where D_k is higher than both D_i and D_j are ruled out. This leads to four different permutations of D_i, D_j and D_k where D_i, D_j and D_k are distinct. We also must consider cases when the D_i, D_j

and D_k are not all distinct. This leads to a total of 6 legitimate cases where at least two of the data segments are merged. Therefore there are a total of 10 different relationships among D_i, D_j and D_k that must be examined.

For each case, we must further consider the following three scenarios that lead to $t_2 \rightarrow t_1$:

(1) t_2 reads a data element $d \in D_k$ written (created) by t_1.

(2) t_2 writes (i.e., creates a new version d^v of) a data element $d \in D_k$ whose predecessor d^o was read by t_1.

(3) t_2 writes (creates a new version d^v of) a data element $d \in D_k$ whose predecessor d^o was written (created) by t_1 and the version created by t2 is read by another transaction.

Now we consider the following 10 cases:

(1) $D_i \uparrow\!> D_j \uparrow\!> D_k$. In this case, obeying HTS leads to the same assertion for all three scenarios: $B_j^k(I(t_2)) > B_i^k(I(t_1))$. From this and the properties 2.1 - 2.3 we can deduce the following: $A_j^i(I(t_2)) \geq A_j^i(A_k^j(B_j^k(I(t_2)) - \epsilon)) = A_k^i(B_j^k(I(t_2)) - \epsilon) \geq A_k^i(B_i^k(I(t_1)) > I(t_1)$. Therefore $t_2 \Rightarrow t_1$.

(2) $D_i \uparrow\!> D_j = D_k$. In this case, obeying HTS leads to the same assertion for all three scenarios: $I(t_2) > B_i^k(I(t_1))$. From this and the properties 2.1 - 2.3 we deduce the following: $A_j^i(I(t_2)) \geq A_k^i(B_i^k(I(t_1)) > I(t_1)$. Therefore $t_2 \Rightarrow t_1$.

(3) $D_i = D_j \uparrow\!> D_k$. In this case, obeying HTS leads to the same assertion for all three scenarios: $B_j^k(I(t_2)) > B_i^k(I(t_1))$. From this and the properties 2.1 - 2.3 we deduce the following: $I(t_2) \geq A_k^i(B_j^k(I(t_2)) - \epsilon) \geq A_k^i(B_i^k(I(t_1)) > I(t_1)$. Therefore $t_2 \Rightarrow t_1$.

(4) $D_i = D_j = D_k$. In this case, obeying HTS leads to the same assertion for all three scenarios: $I(t_2) > I(t_1)$. Therefore $t_2 \Rightarrow t_1$.

(5) $D_j \uparrow\!> D_i \uparrow\!> D_k$. In this case, obeying HTS leads to the same assertion for all three scenarios: $B_j^k(I(t_2)) > B_i^k(I(t_1))$. From this and the properties 2.1 - 2.3 we deduce the following: $I(t_2) > A_k^j(B_j^k(I(t_2)) - \epsilon)) \geq A_k^j(B_i^k(I(t_1))) = A_i^j(A_k^i(B_i^k(I(t_1)))) \geq A_i^j(I(t_1))$. Therefore $t_2 \Rightarrow t_1$.

(6) $D_j \uparrow\!> D_i = D_k$. In this case, obeying HTS leads to the same assertion for all three scenarios: $B_j^k(I(t_2)) > I(t_1)$. From this and the properties 2.1 - 2.3 we deduce the following: $I(t_2) > A_k^j(B_j^k(I(t_2)) - \epsilon)) \geq A_k^j(I(t_1)) = A_i^j(I(t_1))$. Therefore $t_2 \Rightarrow t_1$.

(7) $D_i \uparrow\!> D_k \uparrow\!> D_j$. In this case, the only possible scenario that applies is scenario (1). Under this scenario, obeying HTS leads to the following assertion: $A_j^k(I(t_2)) > B_i^k(I(t_1))$. From this and the properties 2.1 - 2.3 we deduce the following: $A_j^i(I(t_2)) = A_k^i(A_j^i(I(t_2))) \geq A_k^i(B_i^k(I(t_1))) > I(t_1)$. Therefore $t_2 \Rightarrow t_1$.

(8) $D_i = D_k \uparrow\!> D_j$. In this case, the only possible scenario that applies is scenario (1). Under this scenario, obeying HTS leads to the following assertion: $A_j^k(I(t_2)) > I(t_1)$. Therefore $t_2 \Rightarrow t_1$.

(9) $D_j \uparrow\!> D_k \uparrow\!> D_i$. In this case, only scenario (2) applies. There are two sub-cases:

(9.1) d already exists when t_1 read d^o. By HTS protocols this means $B_j^i(I(t_2)) \geq A_i^k(I(t_1))$. However, given that no two timestamps are identical and $B_j^i(I(t_2))$ is a commit timestamp and $A_i^k(I(t_1))$ is an initiation timestamp, $B_j^k(I(t_2)) \neq A_i^k(I(t_1))$. Therefore we have $B_j^i(I(t_2)) > A_i^k(I(t_1))$. From this and the properties 2.1 - 2.3 we can deduce the following: $I(t_2) > A_k^j(B_j^k(I(t_2)) - \epsilon) \geq A_k^j(A_i^k(I(t_1))) = A_i^j(I(t_1))$. Therefore $t_2 \Rightarrow t_1$.

(9.2) d does not exist when t_1 reads d^o. Let the time when t_2 writes be denoted as time2 and the time when t1 reads be denoted as time1, then we have the following: $B_j^i(I(t_2)) \geq C(t_2) >$ time2 > time1 $> I(t_1) \geq A_i^k(I(t_1))$. Therefore we obtain $B_j^i(I(t_2)) > A_i^k(I(t_1))$. By similar reasoning we conclude that $t_2 \Rightarrow t_1$.

(10) $D_j \uparrow\!> D_k = D_i$. In this case, only scenario (2) applies. There are also two sub-cases:

(10.1) d already exists when t_1 read d^o. By HTS protocols this means $B_j^i(I(t_2)) \geq A_i^k(I(t_1))$. By similar reasoning as above we have $B_j^i(I(t_2)) > I(t_1)$. From this and the properties 2.1 - 2.3 we deduce the following: $I(t_2) > A_k^j(B_j^k(I(t_2)) - \epsilon) \geq A_k^j(I(t_1)) = A_i^j(I(t_1))$. Therefore $t_2 \Rightarrow t_1$.

(10.2) d does not exist when t_1 reads d^o. By similar reasoning as (9.2) we conclude $t_2 \Rightarrow t_1$. then we have the following: $B_j^i(I(t_2)) \geq C(t_2) > time2 > time1 > I(t_1) \geq A_i^k(I(t_1))$. Therefore we obtain $B_j^i(I(t_2)) > A_i^k(I(t_1))$. By the same reasoning as above we conclude that $t_2 \Rightarrow t_1$. *Q.E.D.*

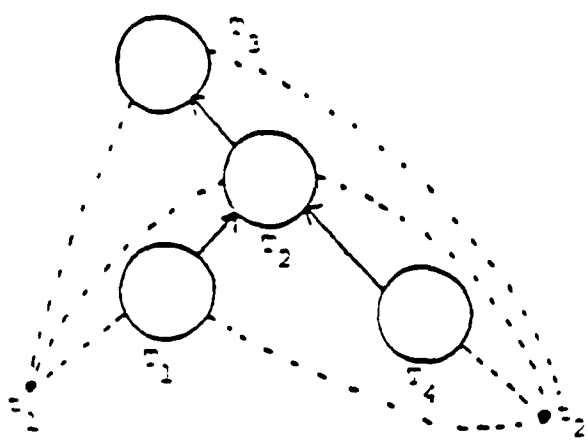
Theorem 2 concludes that the hierarchical timestamp protocols under cyclic partitions enforce the partition synchronization rule. This, combined with the result of Theorem 1, completes the proof of correctness of the algorithm.

6.0 SYNCHRONIZING READ-ONLY TRANSACTIONS

What has been discussed is the algorithm for controlling concurrent update transactions. Now we turn to the read-only transactions.

For a read-only transaction t that reads from data segments that lie on one critical path CP_i^j of the DSH*, the protocol that should be observed is the same as that observed by the update transactions rooted in a data segment immediately below the lowest data segment on the critical path CP_i^j in DSH*, namely, the data segment right below D_i. (If there exists no data segments below D_i in DSH*, then a fictitious data segment can be created to 'host' this read-only transaction.) Therefore read-only transactions will have to obey protocol H alone and will not cause any read timestamp or read lock to be generated. This is graphically presented by transaction t_1 in Figure 15.

What we are concerned with here are those read-only transactions that read from any combination of data segments that do not lie on a critical path in the data segment hierarchy, DHS*, as illustrated by transaction t_2 in Figure 15. In general, what we are looking for is a way to prove the existence of and to derive a consistent database state across all data segments in a database in which the hierarchical decomposition approach to concurrency contorl is used. In other words, if a consist-



•..... t reads from class
(t is a read-only transaction)

Figure 15. Read-only transactions that read from one critical path.

ent database state can be derived, then it can be read by read-only transactions without violating serializability. To achieve this, we first introduce the *extended activity link function* in the following subsection, and prove certain properties of this function that would enable us to derive a consistent database state.

6.1 BASIC DEFINITIONS AND PROOF OF PROPERTIES

In the previous section we have introduced the activity link function which centers around the linkage between transactions rooted in data segments that are on a critical path in the data segment hierarchy. The extended function, on the other hand, specifies how transactions rooted in a data segment are linked to transactions rooted in another data segment when there is not necessarily any critical path in DSH* that connects the two. This function is used to provide a way of computing a *consistent database state* that can be accessed by a read only transaction that reads from any combination of data segments in the database.

The extended activity link function is defined by the functions introduced in the previous chapters, namely, functions A and B. Its usefulness will be indicated in a lemma that follows. The existence and derivation of a consistent database state is given in theorem 3, which makes use of the extended activity link function.

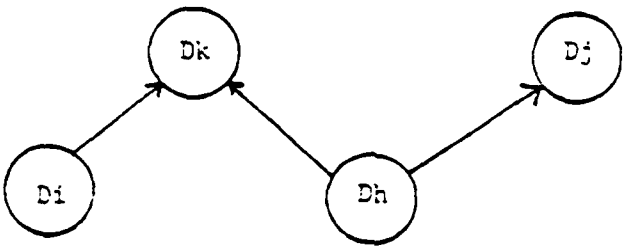
Definition. An *undirected critical path*, denoted as UCP_i^j, is an ordered set of *distinct* indices of data segments in DSH*, such that UCP_i^j = <i, i1, i2, ..., in, j> where for any two indices h, k adjacent in the set, either $D_h \rightarrow D_k$ or $D_k \rightarrow D_h$ is a crtical arc in DSH*.

It is obvious that for any data segment hierarchy DSH* there exists one and only one UCP in DSH* between any pair of data segments. While the activity link function A is defined for a pair of data segments that lie on a critical path, the extended activity link function, using the concept of UCP, is defined for any pair of data segments.

Definition. The *extended activity link function* defined for a pair of data segments D_i and D_j, denoted as $E_i^j(m)$, is a function which maps a time value m to another such that

$$E_i^j(m) = \begin{cases} m \text{ if } i = j, \\ C_i^{late}(m) \text{ if } i \neq j \text{ and } D_j \rightarrow D_i \text{ is a critical arc in DSH*}, \\ I_j^{old}(m) \text{ if } i \neq j \text{ and } D_i \rightarrow D_j \text{ is a critical arc in DSH*}, \\ E_k^j(E_i^k(m)) \text{ otherwise, where } <i,k,\ldots,j> = UCP_i^j, \end{cases}$$

As an example, suppose D_i, D_j, D_k and D_h are in DSH* such that $D_i \rightarrow D_k$, $D_h \rightarrow D_k$ and $D_h \rightarrow D_j$, but D_i and $D_j$are not related in DSH*, as shown in Figure 16. Then UCP_i^j = <i,k,h,j> and $E_i^j(m)$ is evaluated as follows. Take the first pair of indices <i,k> in UCP_i^j and examine the relationship between D_i and D_k. Since $D_i \rightarrow D_k$, therefore I_k^{old} is



$$UCP_i^j = \langle i, k, h, j \rangle$$

$$E_i^j(m) = E_k^j(I_k^{old}(m)) = E_k^j(m')$$

$$E_k^j(m') = E_h^j(C_k^{late}(m')) = E_h^j(m'')$$

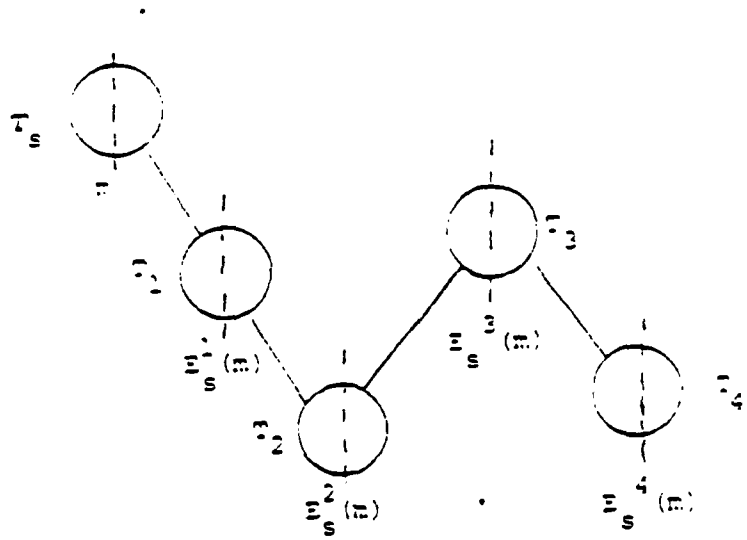
$$E_h^j(m'') = I_j^{old}(m'')$$

Figure 16. An illustration of Evaluation of E function.

invoked and $E_i^j(m)$ is expanded to be $E_k^j(I_k^{old}(m)) = E_k^j(m')$. Then examine the relationship between the next pair of data segments D_k and D_h. Since $D_h \rightarrow D_k$, therefore C_k^{late} is applicable and $E_k^j(m')$ is expanded to be $E_h^j(C_k^{late}(m')) = E_h^j(m'')$. Finally, the relationship between the last pair of data segments on UCP_i^j is examined and since $D_h \rightarrow D_j$, therefore I_j^{old} is applicable and $E_k^j(m'')$ is equated to $I_j^{old}(m'')$. In sum, $E_i^j(m)$ is expanded to be $I_j^{old}(C_k^{late}(I_k^{old}(m)))$. This is illustrated in Figure 16.

The usefulness of the extended activity link function lies in the fact that it can be used to compute the components of a time wall. Intuitively, a *time wall* for all data segments in the database system is a set of times such that no direct dependency from the 'older side' of the wall to the 'newer side' of the wall can occur. A time wall TW(m,s) is composed of an ordered set of all time values, the i-th component of which is expressed as $E_s^i(m)$, where m is a time, D_s is a chosen data segment, and D_i is any data segment. The function E has the property that, given any pair of data segments D_i and D_j, if $t_1 \in D_i$ and t_1 starts before the i-th component of a time wall, and $t_2 \in D_j$ and t_2 starts after the i-th component of the time wall, then t_1 cannot directly depend on t_2 This property is formalized in the following lemma while the concept of a time wall is graphically presented in Figure 17.

Lemma 3.1. Let D_k, D_i and D_j be data segments in a DSH* of a database partition, and D_i and D_j are on one critical path in DSH*. Then



A time wall TW(s,m) is such that no direct dependencies occur between a transaction on the left side of the dotted line (i.e., $T(t) < E_s^s(m)$) and that on the right side of the dotted line (i.e., $T(t) > E_s^s(m)$.)

Figure 17. The E function used as a 'time wall.'

for any time value m and $t_1 \in D_i$, $t_2 \in D_j$, if $I(t_1) < E_k^i(m)$ and $I(t_2) \geq E_k^j(m)$ then there exists no $t_1 \rightarrow t_2$ in the transaction dependency graph $TG(T^u)$ where the concurrency control algorithm enforces the partition synchronization rule.

Proof. Let D_{k1} be the class such that k1 is the fisrt index in UCP_k^i where D_{k1} and D_i, D_j are on one critical path. (k and k1 are not necessarily distict.) Then k1 will also be the first such index in UCP_k^j, and the subset of the ordered set UCP_k^i up to k1 and that of UCP_k^j up to k1 are equivalent. (This is because between any pair of data segments there is one and only one UCP.) Consider the following four groups of cases:

(1) $i = j \neq k1$ or $i = j = k1$. In this case, $E_k^i(m) = E_k^j(m)$. Since t_1 and t_2 are in the same class, by intra-class-synchronization rule we have $I(t_1) < I(t_2)$, which implies that there exists no $t_1 \rightarrow t_2$.

(2) $i = k1 \neq j$. Two cases are considered:

(2.1) $D_i \uparrow\!> D_j$. $I(t_2) \geq E_k^j(m)$ implies that $A_j^i(I(t_2)) \geq A_j^i(E_k^j(m)) = A_j^{k1}(B_{k1}^j(E_k^{k1}(m)))$. From Property 2.1 we have $A_j^{k1}(B_{k1}^j(E_k^{k1}(m))) \geq E_k^{k1}(m) = E_k^i(m) > I(t_1)$. Therefore $A_j^i(I(t_2)) > I(t_1)$, which implies that there exists no $t_1 \rightarrow t_2$.

(2.2) $D_j \uparrow\!> D_i$. $I(t_1) < E_k^i(m)$ implies $A_i^j(I(t_1)) \leq A_i^j(E_k^i(m)) = E_k^j(m) \leq I(t_2)$. Therefore $A_i^j(I(t_1)) \leq I(t_2)$, which implies that there exists no $t_1 \rightarrow t_2$.

(3) $j = k1 \neq i$. Two cases are considered:

(3.1) $D_i \uparrow\!> D_j$. $I(t_2) \geq E_k^j(m)$ implies that $A_j^i(I(t_2)) \geq A_j^i(E_k^j(m)) = E_k^i(m) > I(t_1)$. Therefore $A_j^i(I(t_2)) > I(t_1)$, which implies that there exists no $t_1 \rightarrow t_2$.

(3.2) $D_j \uparrow\!> D_i$. $I(t_1) < E_k^i(m)$ implies $I(t_1) \leq E_k^i(m) - \epsilon$, which implies $A_i^j(I(t_1)) \leq A_i^j(B_j^i(E_k^j(m)) - \epsilon)$. From property 2.2 we have $A_i^j(B_j^i(E_k^j(m)) - \epsilon) < E_k^j(m)$. Since $I(t_2) \geq E_k^j(m)$, therefore $A_i^j(I(t_1)) < I(t_2)$, which implies that there exists no $t_1 \rightarrow t_2$.

(4) $i \neq j \neq k1$. Six cases are considered:

(4.1) $D_i \uparrow\!> D_{k1} \uparrow\!> D_j$. $I(t_2) \geq E_k^j(m)$ implies that $A_j^i(I(t_2)) \geq A_j^i(E_k^j(m)) = A_j^i(B_{k1}^j(E_k^{k1}(m))) = A_{k1}^i(A_j^{k1}(B_{k1}^j(E_k^{k1}(m))))$. By property 2.1, $A_{k1}^i(A_j^{k1}(B_{k1}^j(E_k^{k1}(m))) \geq A_{k1}^i(E_k^{k1}(m)) = E_k^i(m)$. Since $E_k^i(m) > I(t_1)$, we have $A_j^i(I(t_2)) > I(t_1)$, which implies that there exists no $t_1 \rightarrow t_2$.

(4.2) $D_j \uparrow\!> D_{k1} \uparrow\!> D_i$. $I(t_1) < E_k^i(m)$ implies $I(t_1) < E_k^i(m) - \epsilon$, which implies $A_i^j(I(t_1)) \leq A_i^j(E_k^i(m) - \epsilon) = A_{k1}^j(A_i^{k1}(B_{k1}^i(E_k^{k1}(m)) - \epsilon))$. Let $m' = A_i^{k1}(B_{k1}^i(E_k^{k1}(m)) - \epsilon)$. By property 2.2 we have $m' < E_k^{k1}(m)$. Therefore $A_{k1}^j(A_i^{k1}(B_{k1}^i(E_k^{k1}(m)) - \epsilon)) = A_{k1}^j(m') \leq A_{k1}^j(E_k^{k1}(m)) = E_k^j(m) \leq I(t_2)$. Therefore $A_i^j(I(t_1)) \leq I(t_2)$ which implies there exists no $t_1 \rightarrow t_2$.

(4.3) $D_{k1} > D_i > D_j$. $I(t_2) \geq E_k^j(m)$ implies $A_j^i(I(t_2)) \geq A_j^i(E_k^j(m)) = A_j^i(B_i^j(E_k^i(m))) \geq E_k^i(m) > I(t_1)$. Therefore $A_j^i(I(t_2)) > I(t_1)$, which means that there exists no $t_1 \rightarrow t_2$.

(4.4) $D_j \uparrow\!> D_i \uparrow\!> D_{k+1}$. $I(t_1) < E_k^{\,i}(m)$ implies $A_i^{\,j}(I(t_1)) \leq A_i^{\,j}(E_k^{\,i}(m)) = A_i^{\,j}(A_{k+1}^{\,i}(E_k^{\,k+1}(m))) = A_{k+1}^{\,j}(E_k^{\,k+1}(m)) = E_k^{\,j}(m) \leq I(t_2)$. That is, $A_i^{\,j}(I(t_1)) \leq I(t_2)$ which means there exists no $t_1 \rightarrow t_2$.

(4.5) $D_{k+1} \uparrow\!> D_j \uparrow\!> D_i$. $I(t_1) < E_k^{\,i}(m)$ implies $I(t_1) \leq E_k^{\,i}(m) - \epsilon$, which means $A_i^{\,j}(I(t_1)) \leq A_i^{\,j}(E_k^{\,i}(m) - \epsilon) = A_i^{\,j}(B_j^{\,i}(E_k^{\,j}(m)) - \epsilon) < E_k^{\,j}(m) \leq I(t_2)$. That is, $A_i^{\,j}(I(t_1)) < I(t_2)$, which means that there exists no $t_1 \rightarrow t_2$.

(4.6) $D_i \uparrow\!> D_j \uparrow\!> D_{k+1}$. $I(t_2) \geq E_k^{\,j}(m)$ implies $A_j^{\,i}(I(t_2)) \geq A_j^{\,i}(E_k^{\,j}(m)) = A_j^{\,i}(A_{k+1}^{\,j}(E_k^{\,k+1}(m))) = E_k^{\,i}(m) > I(t_1)$. That is $A_j^{\,i}(I(t_2)) > I(t_1)$, which means that there exists no $t_1 \rightarrow t_2$.

For each of the group above we have permutated the level of the distinct classes and for a total of 11 cases we have shown that it is impossible to have $t_1 \rightarrow t_2$. Therefore we prove that there exists no $t_1 \rightarrow t_2$. *Q.E.D.*

The significance of this time wall concept is that if a read-only transaction reads the latest versions of data granules of data segment D_i which are right before the time indicated by the time wall component $E_s^{\,i}(m)$ of certain time wall TW(m,s), then it is accessing a consistent database state and will in no way induce cycles into the transaction dependency graph. This discussion is formally presented in the following theorem.

Theorem 3. If the schedule S enforces the PSR on T^u, and for every d ϵ D_1 that a read-only transaction t_R reads, S allows it to read the version d^o such that

$$TS(d^o) = Max\ (TS(d^v))\ where\ TS(d^v) < E_s^{-1}(m),$$

for some time m and some transaction class index s, then $TG(S(T^u \cup t_R))$ has no cycle.

Proof. In order to prove Theorem 3, we first give the following definitions and a lemma (Lemma 3.2.)

Definition. A *consistent transaction set with respect to a schedule S* of a set of transactions T, abbreviated as a CS w.r.t. S(T), is a set of transactions $T^{cs} \subset T$ such that if $t \in T^{cs}$ and if there exists $t_1 \in T$ such that $t \rightarrow \ldots \rightarrow t_1 \subset TG(S)$, (i.e., if t depends on t_1 in the transitive closure of $\rightarrow$), then $t_1 \in T^{cs}$.

Property 3.1. (The Property of a consistent transaction set.) Partition T^u into T^{u1} and T^{u2}. Then T^{u1} is a consistent transaction set w.r.t $S(T^u)$ iff for any two transactions t_1, t_2, such that $t_1 \in T^{u1}$ and $t_2 \in T^{u2}$, there exists no $t_1 \rightarrow t_2$ in the transaction dependency graph TG(S).

Proof. We want to show that the following two parts are true:

(1) If T^{u1} is a CS then there exists no $t_1 \rightarrow t_2$.

By definition of a CS, if $t_1 \in T^{u1}$ and $t_1 \rightarrow t_2$, then t_2 must be also in T^{u1}, which contradicts the given. Therefore there exists no $t_1 \rightarrow t_2$.

(2) If there exists no $t_1 \rightarrow t_2$ for any $t_1 \in T^{u1}$ and $t_2 \in T^{u2}$, then T^{u1} is a CS.

T^{u1} is a CS because no transaction in T^{u1} can have a dependency in the transitive closure on a transaction which is not in T^{u1}.

Therefore we conclude that this property is true. *Q.E.D.*

Definition. Given a time value m and a *starting data segment* D_s, a *designated consistent transaction set*, denoted as $T^{cs}(m,s)$, is a consistent transaction set such that for all $t \in D_s$, $t \in T^{cs}(m,s)$ iff $I(t) < m$.

Lemma 3.2. Parition T^u into T^{u1} and T^{u2}. Then T^{u1} is the designated consistent transaction set $T^{cs}(m,s)$ w.r.t. $S(T^u)$, where the schedule S enforces the PSR, if T^{u1} contains, for all i, all and only transactions t such that $I(t) < E_s^i(m)$ where $t \in D_i$.

Proof. Construct a time wall TW(m,s). Then by the previous lemma (Lemma 3.1) we know that for any j, k, if $t_1 \in D_j$ and $I(t_1) <$ and $t_2 \in D_k$ and $I(t_2) \geq E_s^k(m)$ then there exists no $t_1 \rightarrow t_2$. Therefore by Property 3.1 above we know that T^{u1} is a consistent transaction set if it contains for all i only transactions t such that $I(t) < E_s^i(m)$ where $t \in D_i$. And since $E_s^s(m) = m$, we have $I(t_1) < m$ if $t_1 \in D_s$. Therefore T^{u1} must be the designated consistent transaction set $T^{cs}(m,s)$. *Q.E.D.*

Corollary. Given a time value m and a starting transaction class T_s, there exists a designated consistent transaction set $T^{cs}(m,s)$.

Theorem 3.

Proof. Partition T^u into T^{u1} and T^{u2} such that for all $t \in D_i$, for all i, $t \in T^{u1}$ iff $I(t) < E_s^i(m)$. Then it is clear that dependencies induced by t_R must be arcs that go from t_R to transactions in T^{u1}, and arcs from transactions in T^{u2} to t_R. By Lemma 3.1, there exist no dependencies from transactions in T^{u1} to those in T^{u2}. Therefore arcs introduced by t_R will not introduce any cycle into the original TG(S). Since TG(S) has no cycle, therefore $TG(S(T^u \cup t_R))$ has no cycle. *Q.E.D.*

In other words, if a read-only transaction reads the latest versions of data granules of data segment D_i which are right before the time indicated by the time wall component $E_s^i(m)$ of certain time wall TW(m,s), then it is accessing a consistent database state and will not induce cycles into the transaction dependency graph.

6.2 CONCURRENCY CONTROL PROTOCOL FOR READ-ONLY TRANSACTIONS

Making use of Theorem 3, a read-only transaction t that reads from data segments that do not lie on one critical path in DHG should be given versions that are the latest before certain time wall. However, to compute the time wall the system has to determine the starting data seg-

ment D_s and a starting time value m. While the choice can be arbitrary, it is theoretically desirable that the following criteria are met:

(1) $E_s^i(m)$ (for all D_i in the DSH*) is computable at I(t), the initiation time of the read-only transaction.

(2) There exists no m' > m such that $E_s^i(m')$ is computable at I(t) for all D_i in the DSH*.

The first criterion stipulates that m should be *small enough* so that all $E_s^i(m)$ is computable at I(t), therefore t potentially does not have to wait until a later time to access from certain segment. (If some $E_s^j(m)$ is not computable at I(t), t would have to wait till a later time when it is computable before accessing data from data segment D_j.) The second criterion strives to achieve reading of the *newest possible* database state.

A compromise is struck here in devising our protocol for read-only transactions. First, to save computation time, a new time wall is computed by the system at certain intervals and the new time wall is 'released' to all read-only transactions that start before the next *version* of the time wall is released by the system. (That is, there is no need to compute a time wall for every read-only transaction.) In computing the next version of the time wall, the system can choose arbitrarily a starting data segment D_s which is of one of the lowest levels and choose m to be the initiation time of the oldest active transaction rooted in D_s. time. If it encounters any C_i^{late} function that it cannot compute,

it waits until it becomes computable. Eventually enough time will elapse such that $E_s^{\,i}(m)$ becomes computable for all D_i's. Then a newly constructed time wall is released.

Let the release time of a time wall TW(m,s) be denoted as RT(TW(m,s)). Now we provide the formal definition of the read-only transaction synchronization protocol.

Concurrency Control Algorithm for Read-Only Transaction

For every database read request from a read-only transaction t for a data granule d, the following protocol is observed:

Protocol R

Let $d \in D_i$. The segment controller of D_i provides the version d^o of d such that

$TS(d^o) = Max(TS(d^v))$ for all v such that

$TS(d^v) < E_s^{\,i}(m)$

where $RT(TW(m,s)) = Max(RT(TW))$ for all TW such that $RT(TW) < I(t)$.

7.0 IMPLEMENTATION OF THE HTS CONCURRENCY CONTROL ALGORITHM

7.1 INTRODUCTION AND SUMMARY OF RESULTS

In this chapter, an implementation of the hierarchical decomposition approach to concurrency control is described. Through this description, the practicality of the proposed algorithm is demonstrated. In our design, we strive to achieve the maximum parallelism at the system level, breaking down the tasks to be performed on the system-level resources into as many parallel units as possible. Techniques described here that pertain to implementation of multi-version databases are also applicable to implementing the conventional multi-version timestamp algorithm. We will also point out the difference in implementation between the HTS concurrency control algorithm and the conventional multi-version timestamp algorithm.

There are two major issues involved in the implementation of the HTS concurrency control algorithm. The first issue is the maintenance of a multi-version database. This is not an issue exclusive to the HTS concurrency control algorithm, but also shared by the conventional MVTS algorithm. It includes problems of how the versions of a data element are to be created, how they are stored and controlled to facilitate rapid accesses, and how they are destroyed to make space for future

versions (i.e., garbage collection). In resolving this issue, an implementation is designed which allows the garbage collection process to operate in parallel with activities that create and access the multi-version database. In addition, we have identified the operations to be performed against a multi-version database using conventional MVTS algorithm to be of two types: ATOMIC_COMMIT and TIMESTAMPED_READ. We show that the result of using the HTS timestamp algorithm is to allow a third type of operations, called the NO_TRACE_READ operations, to replace a certain number of the TIMESTAMPED_READ operations. We will in our description of the implementation demonstrate the fact that the NO_TRACE_READ operation is allowed to proceed without ever having to be blocked, while the TIMESTAMPED_READ operation still faces the danger of being blocked due to contention for system-level resources. This result serves to further substantiate the argument that leaving read timestamps is a relatively expensive operation, in addition to its potential of causing more transaction aborts.

The second issue involves the mechanisms of implementing Protocol L and Protocol H. Recall that in order to use Protocol H to access a data element in a higher data segment, a read time ceiling, computed by evaluating an A function, must be available. On the other hand, in using protocol L to access a data element in a lower data segment, a timestamp for accessing that data segment which is different from the transaction's own initiation timestamp, is used to synchronize such accesses, and the use of which will involve certain constraints to be enforced.

Accomplishing these tasks requires maintaining additional information. How this additional information is created, stored, accessed and destroyed is the main subject in discussing this second issue. In sum, an implementation is identified which enables a rapid computation of the A function and does not require the computation process ever to be blocked due to concurrent accesses to the required information. This feature, along with the fact that multiple uses of the Protocol H by a single transaction to access data elements in the same higher data segment would require the computation of the A function to be performed only once, makes it possible to efficiently implement Protocol H.

The organization of this chapter is as follows. Section 2 provides an overview of the tasks to be performed by the HTS concurrency control mechanism. This overview serves to identify the various operations that will be applied to the multi-version database and other relevant information. Section 3 describes the implementation of the multi-version database and how it is used to support concurrent application of the three types of database operations: ATOMIC_COMMIT, TIMESTAMPED_READ and NO_TRACE_READ. It also describes mechanisms for garbage collection. Section 4 describes the implementation of the transaction history information that is required to facilitate the evaluation of the A function and to enforce constraints necessitated by the psudo evaluation of the B function.

7.2 OVERVIEW OF TASKS OF THE HTS CONCURRENCY CONTROL MECHANISM

In this overview, we will describe the implementation of the HTS concurrency control mechanism from three different angles: (1) The tasks of the mechanism as seen by a transaction (i.e., the interface of the concurrency control facility to a transaction,) (2) The modules of the concurrency control mechanism, and (3) The data (system-level resources or database data) to be accessed by these modules. This description serves to provide a perspective on issues involved in implementing the algorithm and motivates the detail description in the subsequent sections.

7.2.1 INTERFACE OF THE CONCURRENCY CONTROL FACILITY TO TRANSACTIONS

From the point of view of a transaction, interactions with the concurrency control facility take place at three points: initiation, reading database data, and finishing. This is shown in Figure 18. When a transaction is initiated, INITIATION must be called to obtain an initiation timestamp for the transaction. During the execution of the transaction, whenever the transaction requests to read a data element in the database, the read request must be handled through a READ call to the concurrency control facility. However, since every uncommitted transaction is subject to the possibility of user cancellation and system abort, when the transaction performs a write to the database before it is finished, it can not directly write into the database, but should write into its own work space. This is a standard technique used to

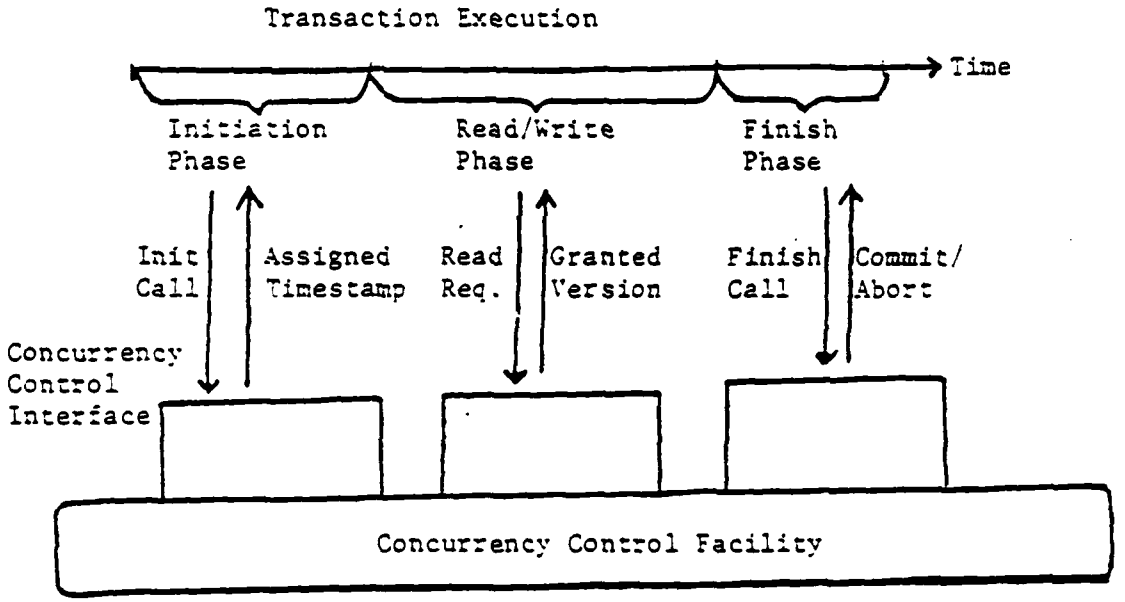


Figure 18. Interface of the concurrency control mechanism to a transaction.

prevent the system from cascading the effect of the transaction to other transactions before it is committed. When a transaction is finished processing, its actual commit or abort is then handled by a FINISH call to the concurrency control facility, which validates all the writes this transaction has performed by comparing the transaction's timestamps with timestamps of data elements in the database. If the transaction passes the validation phase, it is then committed, and all the writes will be performed in the database. Otherwise it is aborted and restarted.

Note that whether these 'calls' are done through message passing or subroutine calls is not relevant for the purpose of our current discussion. We assume that the concurrency control facility is capable of being executed by multiple processes, and the need for mutual exclusion when multiple processes are in session will be handled by accesses to system-level semaphores or locks when it arises. Therefore whether the facility is executed by the processes that also execute the transactions or it is executed only by dedicated processes does not alter the correctness of its execution. The choice would depend strictly on the nature of the processing environment.

Note also that we do not attempt to address separately the issue of crash recovery. Crash recovery can be handled in the same way as it is handled in other concurrency control methods. If the system crashes during execution of a transaction, that transaction is restarted when the system recovers without the database being contaminated. To handle

the problem of system crashes during the commit phase of a transaction, one may resort to the standard technique of two-phase commit <Gray78, Chan82b>. In essence, two-phase commit requires that the commit phase of a transaction be decomposed into two parts: pre-commit and post-commit. During pre-commit, the writes performed by the transaction are forced to a permanent storage (e.g., a log disk). If the system crashes during pre-commit, the transaction is considered uncommitted and therefore restarted when the system recovers. After pre-commit is completed, the transaction is considered committed and undergoes the post-commit phase during which its writes are actually performed in the database. If the system crashes during the post-commit phase of a transaction, the transaction is recovered when the system recovers by performing writes (i.e., redo) into the database based on the log. Since implementation of the HTS concurrency control algorithm does not preclude the use of two-phase commit (i.e, pre-commit can be easily integrated into the validation phase), crash recovery is not specially discussed.

7.2.2 MODULES OF THE CONCURRENCY CONTROL FACILITY

We will now provide an overview of the modules to be executed within each of the three different tasks of the concurrency control facility.

7.2.2.1 INITIATION

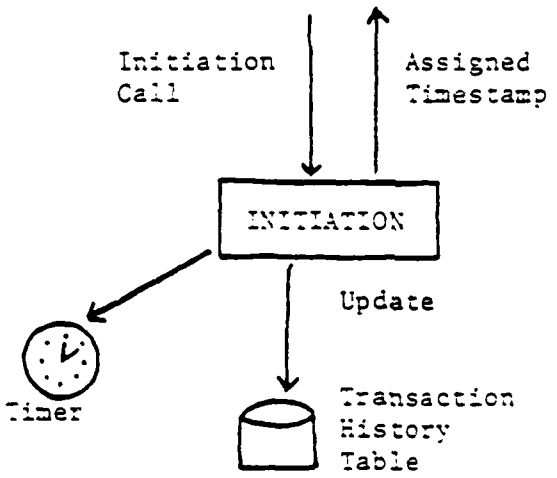


Figure 19. Description of INITIATION.

As shown in Figure 19, INITIATION is fairly simple. It merely reads the timer and returns a value as the assigned timestamp for the transaction. It is assumed that no two readings of the timer will result in the same time value. In addition, INITIATION must also record initiation of new transactions in a transaction history table, which will be used by modules that evaluate the A function.

7.2.2.2 READ

In the conventional MVTS algorithm, the READ call to the CC (concurrency control) facility will result in a TIMESTAMPED_READ operation. This operation is passed with the timestamp of the transaction and an identifier of the data element to be read (e.g., a logical page id), and is expected to return the desired version, or the (virtual) address of the physical page that contains the desired version. TIMESTAMPED_READ performs the following tasks:

(1) Decide which version of the data element is right before the timestamp of the transaction;

(2) Leave the timestamp of the transaction with this version;

(3) Return (the address of) this desired version.

In the HTS timestamp algorithm, READ of the CC facility will first have to determine whether the request is a read to the root data segment of the transaction, to a higher data segment, or to a lower data segment. As shown in Figure 20, three different modules are defined to

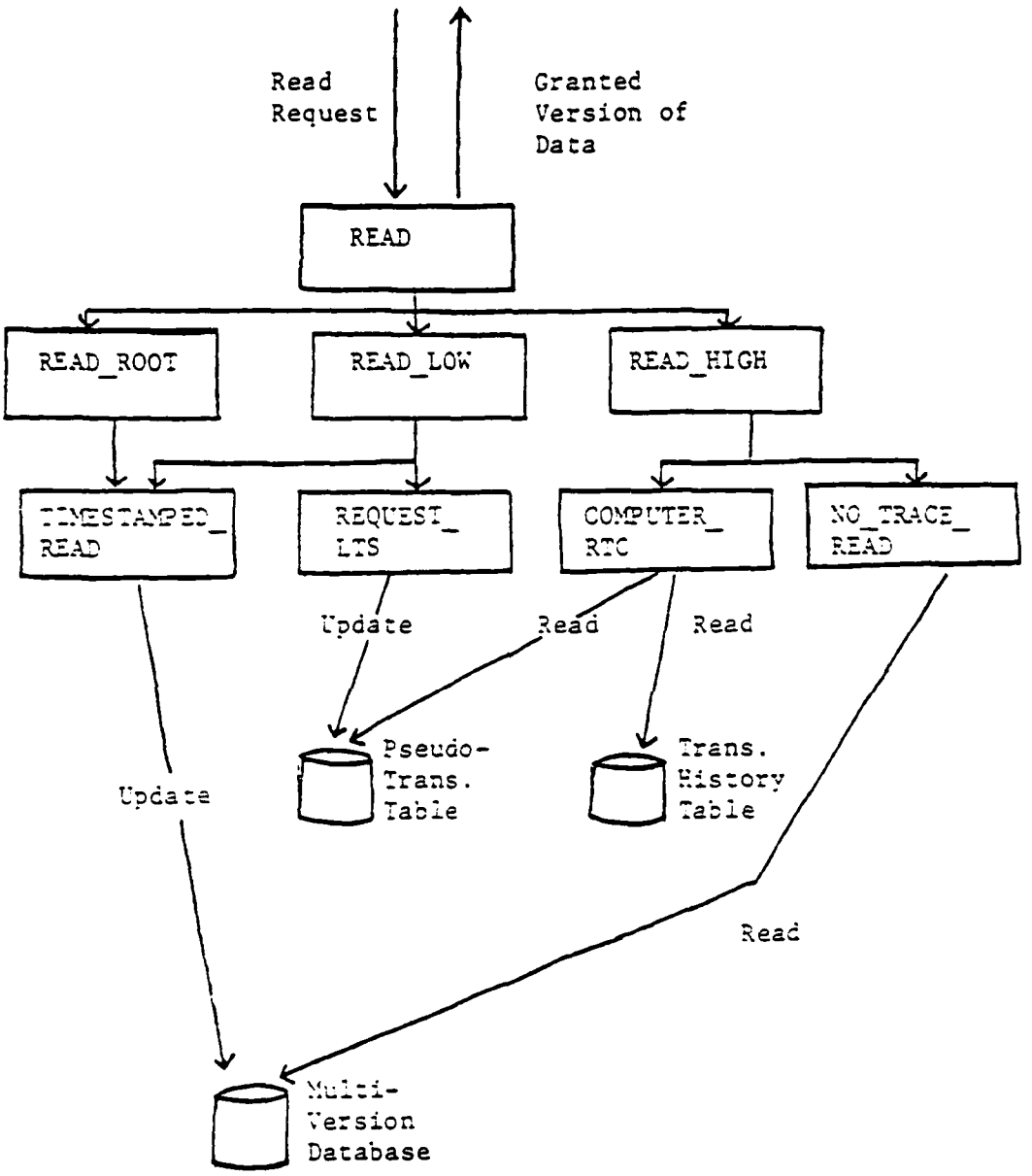


Figure 20. Description of READ.

handle these three cases. READ_ROOT implements the read access of Protocol E, which simply invokes TIMESTAMPED_READ with the transaction timestamp and the data id. READ_HIGH implements the read access part of Protocol H. READ_HIGH contains a sub-module, COMPUTE_RTC, which evaluates the A function given the timestamp of the transaction and the identifiers of the root data segment and the higher data segment being accessed. (RTC stands for read time ceiling.) Once the result of the evaluation is available, READ_HIGH invokes NO_TRACE_READ operation, passing to it the result of the evaluation as the read time ceiling, and the data id. NO_TRACE_READ performs the following tasks:

(1) Decide which version of the data element is right before the read time ceiling;

(2) Return (the address of) the desired version.

Note that the difference between the TIMESTAMPED_READ operation and the NO_TRACE_READ operation is that the latter does not have to leave a read timestamp with the version of the data element being accessed.

The third module, READ_LOW, handles the read access part of Protocol L. It contains a sub-module REQUEST_LTS which provides timestamps that accesses to lower data segments should use for synchronization. (LTS stands for lower timestamp.) This sub-module essentially 'guesses' the value of the B function without actually evaluating it, and maintains constraints to be enforced in order to validate its guesses. (Refer to section 5.2 for details.) The value provided by REQUEST_LTS will be

used as the timestamp value to be passed to TIMESTAMPED_READ, an operation also shared by the READ_ROOT module.

As also shown in Figure 20, three types of data are being maintained and accessed by the modules implementing READ. The transaction history table is read by COMPUTE_RTC to evaluate the A function. The pseudo-transaction table is updated by REQUEST_LTS to record constraints to be enforced as a result of guessing at the values of the B function. This table is also read by COMPUTE_RTC. Finally, the multi-version database is consulted by both TIMESTAMPED_READ and NO_TRACE_READ to determine the correct version to be read and the virtual address of that version.

7.2.2.3 FINISH

The tasks that the HTS algorithm performs when a transaction is finished processing are very similar to those performed by the conventional MVTS algorithm. As shown in Figure 21, FINISH in essence implements the operation ATOMIC_COMMIT. ATOMIC_COMMIT consists of two stages. At the first stage, the module VALIDATION is invoked which, for every data element that the transaction has written, checks to see if the version to be created by this transaction has been invalidated by a read timestamp on the version immediately previous to the version to be created. This validation process will result in the transaction being considered either aborted or committed. In the former case, the second stage of

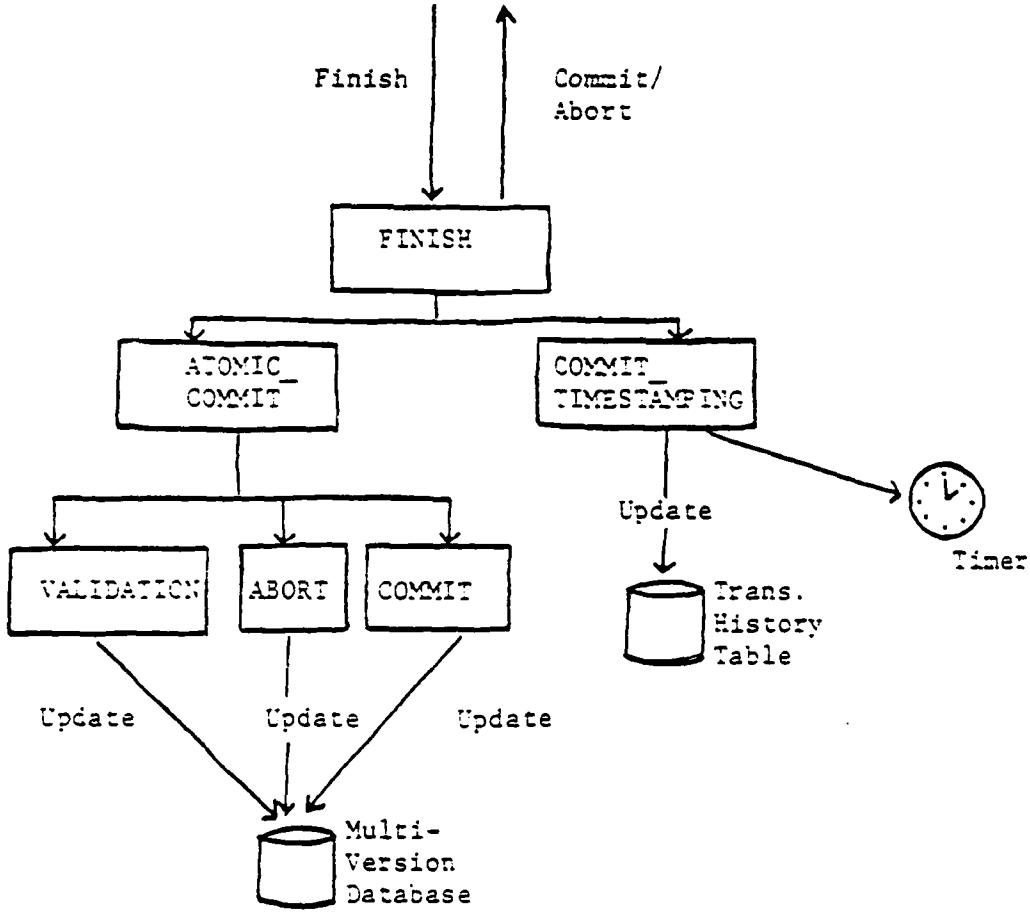


Figure 21. Description of FINISH.

ATOMIC_COMMIT will invoke the module ABORT to cause the writes performed by the transaction to be discarded and the transaction restarted. In the latter case, it will invoke COMMIT to cause new versions to be actually created in the multi-version database. ATOMIC_COMMIT is an operation that has to be accomplished atomically, i.e., it cannot allow writes that have been validated to be invalidated before the operation is finished.

An additional module executed by FINISH which is unique to the HTS algorithm is the COMMIT_TIMESTAMPING module. This module assigns a commit timestamp to the committed transaction and records this information in the transaction history table.

As also shown in Figure 21, the three kinds of data accessed by READ are also used in FINISH. Transaction history data is updated by COMMIT_TIMESTAMPING module to record commit times of transactions. Pseudo-transaction table is consulted by VALIDATION at the end of the validation phase tc make sure that the transaction has not been in progress for too long so as to violate constraints posed by pseudo transactions. And finally the multi-version database is used by all three modules within ATOMIC_COMMIT.

7.2.3 SHARED DATA IN THE CONCURRENCY CONTROL FACILITY

Many modules of the above three components of the concurrency control facility, INITIATION, READ and FINISH, share accesses to the three types of data relevant to the concurrency control facility. In the current sub-section, we will analyze potential concurrent accesses to the shared data resources. This analysis is essential to designing the data structures and the access procedures to these data structures so as to allow for the maximum level of concurrency *within* the concurrency control facility.

From a theoretical point of view, the three tasks of the concurrency control algorithm, INITIATION, READ, and FINISH (i.e., writes) are all atomic tasks. By an atomic task we mean that the correctness of the algorithm relies on the expectation that the (sub-)process that executes the task is not interleaved with any other process. However, since each of these tasks in reality will involve many instructions and execution steps that are potentially lengthy, enforcing atomicity of these tasks by allowing the concurrency control facility to execute one task at a time would inevitably cause the facility to become a bottleneck. Therefore it is important that these tasks be analyzed and their semantics be understood so that a design can be achieved that allows as many concurrent tasks to proceed as possible. This concept of emulating the effect of atomic execution of tasks while allowing multiple tasks to proceed at the same time is analoguous to emulating the effect of atomic execution of transactions while allowing multiple transactions to proceed at the same time. The latter is achieved by the concurrency control facility

of a DBMS, while the former is to be achieved through a careful design of the facility itself.

7.2.3.1 DESIGNING SYSTEM MODULES TO MAXIMIZE CONCURRENCY - A METHODOLOGY

The methodology that we use to achieve a design of the data structures and access procedures that allows the maximum level of concurrency within the concurrency control facility is as follows.

(1) Obtain a list of the types of atomic tasks that will have to be performed against shared data. For each of these types of tasks analyze the semantics of the tasks and embark on an attempt to break the tasks down into work units that must, based on the semantics of the tasks, be executed atomically. These work units, referred to as *atomic work units*, may be organized in a hierarchical fashion, in the sense that one atomic work unit may further contain other atomic work units as atomic sub-units. The criterion for determining whether a work unit is atomic is whether the resources read or written by the work unit can be released when the unit is finished. Therefore if an atomic unit contains other atomic work units as sub-units, then the resources obtained and used by the atomic sub-units may be released when the sub-units are finished, while the resources obtained and used by the parent atomic unit can not be released until the parent unit is finished.

(2) Construct a data-operation matrix where on one dimension the shared data sets are listed and on the other dimension the atomic work units are listed. For each cell in the matrix, the operation that the atomic unit will perform on the data set is listed. The operations could be 'read', 'write' or 'read and write'. The matrix is used to facilitate conflict analysis.

(3) Conduct conflict analysis for each pair of atomic work units to identify potential conflicting atomic work units. Two atomic work units conflict if the intersection of the data sets they access is not empty and at least one of the units would have to perform a write on one of the data sets in the intersection set. The result of conflict analysis is captured in a conflict matrix where each of the dimensions consists of all the atomic work units.

(4) For each pair of conflicting atomic work units, examine the nature of conflict and identify implementation methods that will handle the conflict efficiently.

Three methods for handling conflicts are identified and listed below:

(1) Serialization: Allow only one of the conflicting pair of atomic work units to proceed at any time. This can be achieved through the use of a single dedicated process to handle both types of atomic work units. This method is effective when the atomic work units involved are very short.

(2) Mutual exclusion on selected granules of data sets: Allow multiple work units to proceed at the same time but serialize those that are contending for the same data granules. This is achieved by associating with each granule of data in the data set a semaphore, or a lock. The lock is obtained when the data granule is to be operated on and is released when the atomic unit is finished. This method is more expensive than serialization because it involves overhead of setting, releasing, testing for locks, and blocking. However, it allows a higher level of concurrency.

(3) Semantic analysis for read-write conflict: For those conflicts in which one atomic unit only needs to read the data set, semantic analysis may be combined with careful implementation to enforce that no writes onto the granules will (semantically) invalidate a previous read of the granules by the read-only atomic unit still in progress. This method, if applicable, eliminates the conflict without incurring the overhead of blocking and locking.

7.2.3.2 APPLYING THE METHODOLOGY TO DESIGNING CONCURRENCY CONTROL FACILITY

We will apply the above methodology to the implementation of the HTS control algorithm. The atomic work unit hierarchy, data-operation matrix and the conflict matrix are shown in Figure 22, Figure 23(a) and

Atomic Work Units	Accessed Data Sets	Access Mode
1. INITIATION	Trans. History Table	Update
2. COMMIT_ TIMESTAMPING	Trans. History Table	Update
3. REQUEST_LTS	Pseudo-trans. Table	Update
4. COMPUTE_RTC	Trans. History Table Pseudo-trans. Table	Read
5. NO_TRACE_READ	MVDS	Read
6. TIMESTAMPED_READ	MVDS	Update
7. ATOMIC_COMMIT	MVDS	Update

Figure 22. The Atomic Work Unit Hierarchy of the HTS Concurrency Control Facility

Work / Data Units / Sets	INIT	CTS	RLTS	CRTC	NTR	TSR	AC
MVDB					R	W/R	W/R
PTT			W	R			
THT	W	W		R			

INIT: INITIATION
CTS: COMMIT_TIMESTAMPING
RLTS: REQUEST_LTS
CRTC: COMPUTE_RTC
NTR: NO_TRACE_READ
TSR: TIMESTAMPED_READ
AC: ATOMIC_COMMIT

MVDB: Multi-version database
PTT: Pseudo-trans. table
THT: Trans. history table

W:Write
R:Read

	INIT	CTS	RLTS	CRTC	NTR	TSR	AC
INIT	THT	THT		THT			
CTS		THT		THT			
RLTS			PPT				
CRTC							
NTR						MVDB	MVDB
TSR						MVDB	MVDB
AC							MVDB

Figure 23. The Data-Operation Matrix and the Conflict Matrix of CC Facility.

Figure 23(b). Analying the conflict matrix leads to the following design decisions:

(1) We would like to have multiple NO_TRACE_READ's to proceed without ever having to be blocked or to incur locking overhead. Since its conflict with ATOMIC_COMMIT and TIMESTAMPED_READ is of the read-write type, method (3) listed above is applicable. We will identify a design of the multi-version data base and a procedure for accessing this data set by all three types of atomic work units that would achieve this goal.

(2) We would like to have multiple COMPUTE_RTC's to proceed without ever having to be blocked or to incur locking overhead. Since its conflict with REQUEST_LTS, INITIATION and COMMIT_TIMESTAMPING is of the read-write type, method (3) listed above is again applicable. A design of the transaction history table and the pseudo-transaction table and access procedures to these data sets is identified to achieve this goal.

(3) Since it is believed that INITIATION and COMMIT_TIMESTAMPING are short atomic units, we will use method (1) above to reconcile their conflicts.

(4) All the other conflicts are to be resolved through system-level locks. These locks will be integrated in the design of these data sets.

7.2.4 SUMMARY

In this section we have provided an overview of the tasks of the concurrency control facility that implement the HTS algorithm. The overview focuses on the issue of how to design the implementation in a way so as to maximize concurrency within the facility. A methodology for analyzing the tasks of the facility, which is applicable to the implementation of other facilities in a DBMS or an operating system, is presented and applied to the current problem. Conclusions are drawn to guide the design that are to be presented in the subsequent sections.

7.3 IMPLEMENTATION OF THE MULTI-VERSION DATABASE

7.3.1 BASIC CONCEPTS

In this section we describe the techniques for implementing the multiv-version database to be accessed by the three major atomic work units ATOMIC_COMMIT, TIMESTAPMED_READ and NO_TRACE_READ of the concurrency control facility. (Refer to the conflict matrix shown in Figure 23 on page 141.)

We assume that all the data elements in the database that the processes outside of the concurrency control facility may issue reads and writes to and that are target for control by the CC facility are fixed-length *logical pages*. This assumption, however, does not compromise the generality of the results in our current design. In fact, any

unit of data objects, such as tuples or records of relations or files, can be used as the units of data elements on whose behalf multiple versions are to be kept, so long as their identities can be represented and mapped easily onto a storage unit. (Examples of storage units are virtual addresses and physical pages.)

A logical page is identified by a unique logical page id. Given a database system, a fixed number of logical pages are allocated. All these pages are assumed to contain some information which may be user data, system data or simply information that indicates that the page is available for storing new information. The semantics of the content of the logical pages are of no concern to the concurrency control facility.

The concurrency control facility maintains multiple versions of these logical pages. However, the fact that multiple versions exist for each logical page is transparent to all processes outside of the concurrency control facility. Within the CC facility, versions of logical pages are assigned to physical pages, each of which corresponds to a virtual memory page and is identified by the virtual memory address.

7.3.2 STRATEGIES FOR IMPLEMENTING MULTIPLE VERSIONS

The analysis presented in the previous section shows that within the CC facility two major types of accesses to the MVDB are the following:

(1) Given a logical page id, find the physical page that contains the most recent version of the logical page.

(2) Given a logical page id and a time ceiling, find the physical page that contains the most recent version of the logical page prior to the given time ceiling.

The MVDB must be implemented to facilitate fast responses to these types of accesses. To accomplish the first, we will allow direct accesses to the most recent version of a logical page. To accomplish the second, we will allow the versions of a logical page be chained together in a reversed chronological order so that a request to a version subject to some time ceiling can be achieved through a direct access to the most recent version and then traversing the chain until a version with a version timestamp smaller than the given time ceiling is found. Therefore the guidelines for the implementation are the following:

(1) Direct access to the most recent version of a logical page.

(2) Versions of a logical page are to be chained in a reversed chronological order.

There are two strategies for implementing direct accesses to the most recent versions. The first one is physical clustering, namely, reserving a continuous block of physical pages to store the most recent versions of all the logical pages so that given a logical page id the virtual address of the physical page that stores the most recent version

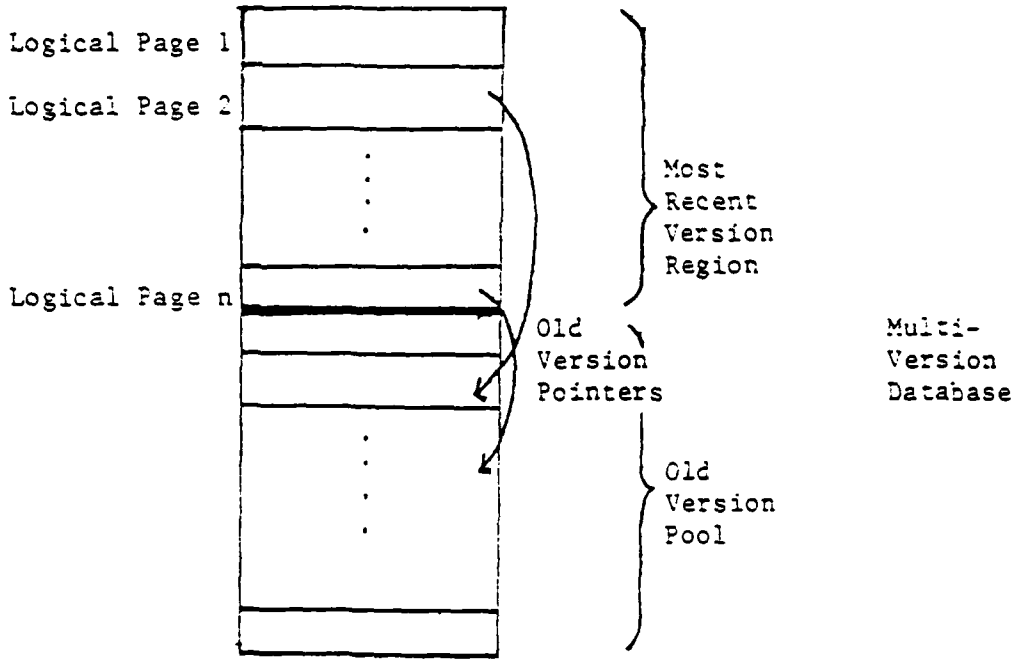


Figure 24. Implementing MVDB using the physical clustering technique.

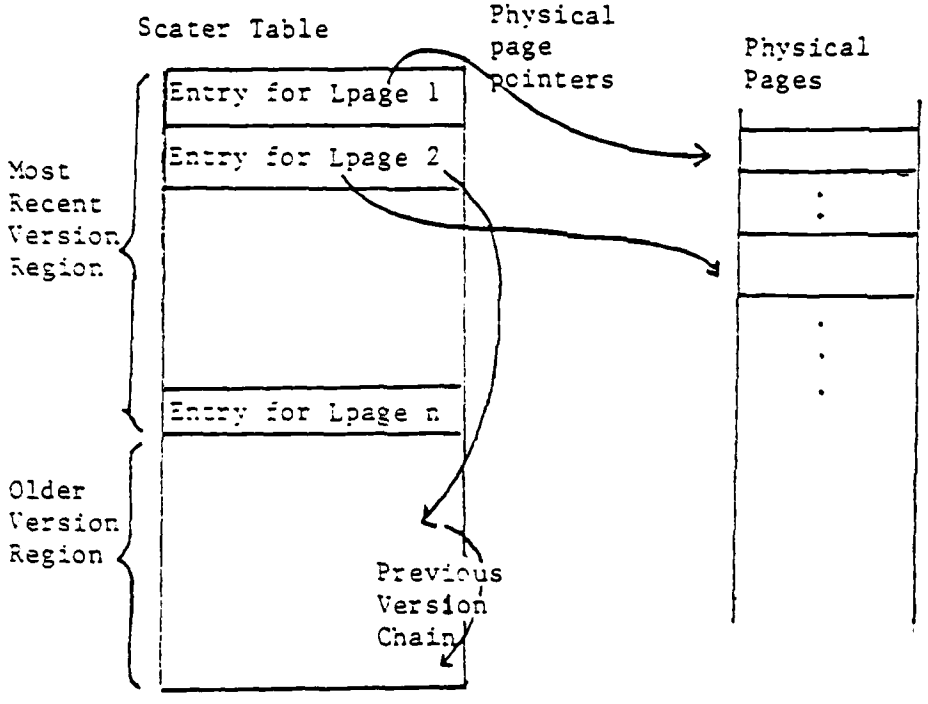


Figure 25. Implementing MVDB using the scatter table technique.

of that logical page can be directly computed from that logical page id. The second one is through the use of a scatter table. Given a logical page id, the address of the entry that corresponds to the logical page in the scatter table is directly computable from the logical page id, while the virtual address of the physical page that contains that version is stored in the scatter table entry. The physical clustering method is shown in Figure 24, and the scatter table method is shown in Figure 25.

The advantages and disadvantages of using the scatter table method versus the physical clustering method are listed below:

Advantages:

(1) Avoid the need of copying versions multiple times: The need for atomic commit does not allow new versions of logical pages created by a transaction (i.e., writes on these logical pages by that transaction) to be made known to the database until it is decided that the transaction will commit. This amounts to requiring that new versions be first created in a region of physical pages assigned to be the transaction's own work space. Under this circumstance, if physical clustering is used to implement the most recent versions, then most of the new versions will have to be written twice every time it is created: once in the private work space of the transaction that creates it and once in the most recent versions' cluster region. Moreover, when a version is superceded by a more recent version, it

has to be forced out of the most recent versions' cluster region and be copied into an area for older versions. However, if the scatter table method is used, the physical pages in the private work space where the new versions were first created can be directly pointed to by the scatter table when the transaction commits, thereby avoiding the overhead of writing the new versions twice. Similar arguments persist when a version is superceded and must be forced out of the most recent versions' region.

(2) Higher level of spatial locality when traversing the version chain: Since scatter table entries are likely to be much smaller in size than the logical or physical pages, less space (i.e., less number of physical pages) is required to complete the paths of version chains. Traversing a version chain to find a desired version can be accomplished by traversing scatter table entries rather than by actually visiting individual physical pages that store the intermediate versions of the logical page. Spatial locality is therefore likely to be enhanced during activities of version chain traversals. (Higher spatial locality is desirable because it potentially enhances the performance of the virtual storage system.)

Disadvantages:

(1) Avoid the overhad of maintaining and backing up the scatter table.

(2) Higher level of spatial locality in the most recent versions' region: Since the most recent versions are more likely to be the desired versions that satisfy most access requests, and assuming that spatial locality exists at the logical page level (i.e., contiguous logical pages have a higher probability of being accessed together,) it is expected that spatial locality at the physical page level can be enhanced if the most recent versions are also physically clustered.

In our current design, we have chosen to use the scatter table technique for implementation. The remainder of this section describes how operations on the multi-version database are accomplished in this design. However, this choice does not compromise the generality of the main results we will obtain for the implementation. A design that has similar attributes in conflict handling but uses the physical clustering technique can be analoguously obtained.

7.3.3 DATA STRUCTURES

The data structure of the scatter table is shown in Figure 26. The table is composed of two parts. The first part, called the most recent version region, stores the entries for the most recent versions. The second part, called the old version region, stores the entries for the older versions. The number of entries in the first part is determined by the number of logical pages allocated in the system, while in the

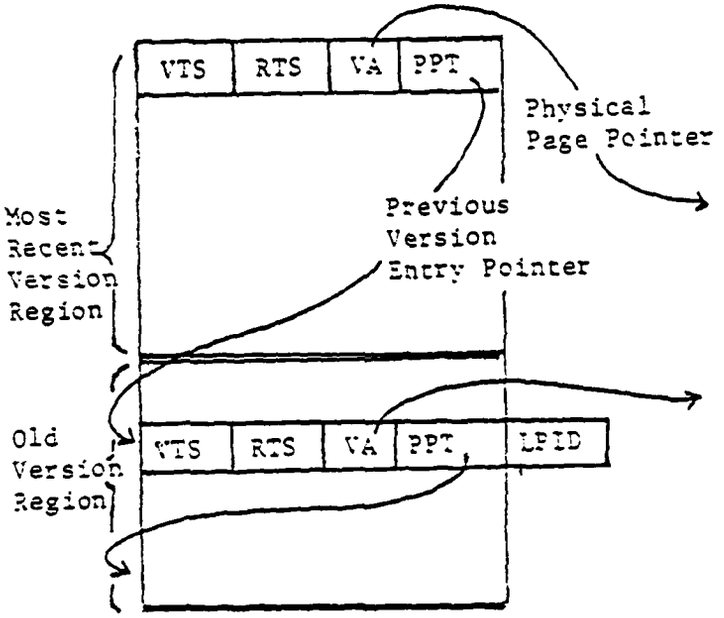


Figure 26. Data structure of the scatter table.

second it is a function of what is considered adequate for holding all the older versions that might still be accessed at any time.

For the purpose of garbage collection, the old version region is arranged as a ring buffer, with two pointers FREE_SPACE_BEGIN and FREE_SPACE_END pointing to the beginning of the free entries and the end of the free entries. When FREE_SPACE_BEGIN catches up with FREE_SPACE_END, the scatter table for the older versions is full and the system must wait till garbage collection to advance the FREE_SPACE_END pointer before it can further allow most recent versions to be superceded. We will design the garbage collection process to be a parallel process that is constantly checking to get rid of stale versions.

As shown in Figure 26, each scatter table entry contains (1) the version timestamp (VTS), i.e., the timestamp used by the transaction to create this version, (2) the read timestamp (RTS), i.e., the largest timestamp of the transactions that read this version, (3) the pointer to the scatter table entry of the immediately previous version (PPT), and (4) the virtual address of the physical page that stores this version (VA). For an entry in the old version region of the scatter table, the logical page id whose old version the entry is corresponding to is also stored with the entry.

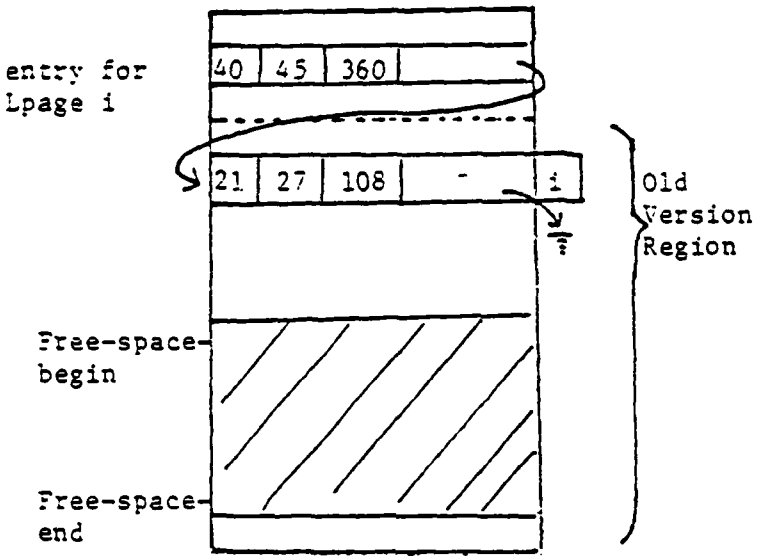
7.3.4 OPERATIONS AND SYNCHRONIZATION

We will now analyze the operations to be performed on the scatter table by the three atomic work units: ATOMIC_COMMIT, TIMESTAMPED_READ and NO_TRACE_READ. The purpose is to identify the most efficient way of controlling concurrent accesses to the scatter table while maintaining the correctness of each operation.

7.3.4.1 ATOMIC_COMMIT

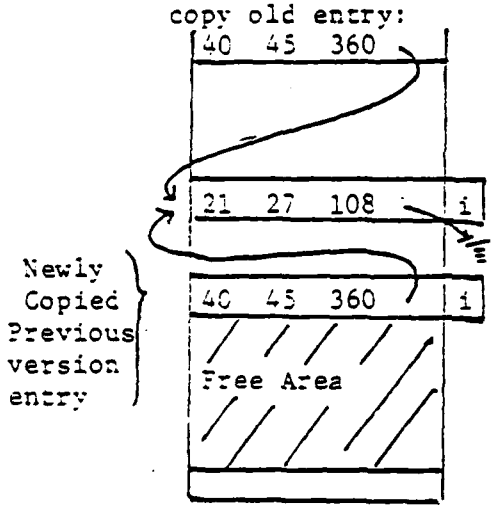
The task of ATOMIC_COMMIT for a transaction with a timestamp TS is as follows. (We assume that, for practicality, only the most recent versions can be superceded by a new version.) For every new version to be created, the scatter table entry for the most recent version is located and the RTS field of that entry is compared with TS and the entry is validated if RTS $\leq$ TS and VTS $\leq$ TS. If any entry cannot be validated, the transaction is aborted. If all entries are validated, then the transaction commits. When the transaction commits, for every new version it creates, a new entry in the scatter table must be created and placed in the proper location in the version chain. This can be accomplished by, for each new version to be created, obtaining a free entry space in the Old Version Region and copying the entry for the immediately previous version to the free entry space. The entry for the new version can then be placed in the entry space originally occupied by the entry for the immediately previous version. This process of inserting entries for new versions in the scatter table is shown in Figure 27.

Original Scatter Table State:



Wishes to insert a new version 66 of Lpage i now in physical page address 395:

Step 1: Obtain a free space from Free Area and copy old entry:



Step 2: Write over the new most recent version

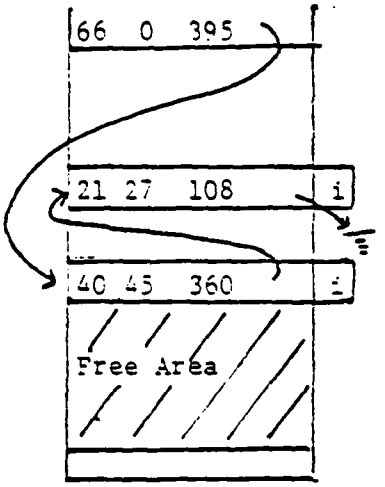


Figure 27. Procedure for inserting entries of new versions into the scatter table

The conflict among ATOMIC_COMMIT processes arise from version chain insertions. If two ATOMIC_COMMIT processes are run in parallel and both of them need to insert a new version of the same logical page then use of the above insertion procedure will lead to inconsistency. The inconsistency is caused by the invalidation of a validated entry by concurrent processes. To prevent this situation, a lock bit, called ENTRY_LOCK, is associated with each entry which must be acquired and locked before an entry is examined for validation. The lock will be held until the transaction is commited and the insertion procedure for that logical page is finished, or until the transaction aborts.

7.3.4.2 TIMESTAMPED_READ

The task for TIMESTAMPED_READ is simpler. Given a read request to a logical page and a timestamp ceiling TS, the version chain in the scatter table for that logical page is traversed until the entry for the version with a version timestamp $\leq$ TS is located. The RTS field of that entry is then updated to be the maximum of its original value and TS. The address (i.e., VA of the entry) is then returned.

The conflict between TIMESTAMPED_READ and ATOMIC_COMMIT lies with accesses to the RTS field of an entry. Updating RTS of an entry already validated by a concurrent ATOMIC_COMMIT process could result in the invalidation of the entry for the ATOMIC_COMMIT process. To prevent such interference, TIMESTAMPED_READ is required to acquire ENTRY_LOCK

associated with an entry before it reads it, and will immediately release it after it is determined that this is not the version needed or, if it is, after RTS is updated.

The conflict among TIMESTAMPED_READ's also arises from conflicting accesses to the RTS field, and therefore can be handled by the above ENTRY_LOCK protocol.

7.3.4.3 NO_TRACE_READ

The task of NO_TRACE_READ is very similar to that of TIMESTAMPED_READ. The only difference is that the former does not require the update of the RTS field of the entry for the version to be read. Since the logic of NO_TRACE_READ is not concerned with RTS, the only conflict is between NO_TRACE_READ and ATOMIC_COMMIT when the latter inserts entries in the version chain that the former is following. However, the following two facts enable NO_TRACE_READ processes to proceed without ever having to be concerned with concurrent ATOMIC_COMMIT processes that operate on the version chain of the same logical page:

(1) If entry e1 is locked by a concurrent ATOMIC_COMMIT process, then the version to be inserted by that process cannot be the target of any concurrent NO_TRACE_READ process. This is because the time ceiling that a NO_TRACE_READ process uses is always smaller than the timestamps of any active transactions that might write on that logical page. (Refer to the definition of Protocol H.)

(2) Reading a locked entry e1 by a NO_TRACE_READ process will not put the NO_TRACE_READ process in danger of reading a distorted version chain which may lead to inconsistency. This can be read from Figure 27 on page 154 which shows the possible states that a NO_TRACE_READ may encounter when retrieving a version of a data element while a concurrent process is inserting a new version. It is shown that NO_TRACE_READ would not be misled even if it reads the intermediate state produced by the concurrent insertion process.

Therefore it is concluded that a NO_TRACE_READ process can proceed without ever interfering with any other concurrent process, and therefore never has to wait or obey other synchronization protocol.

7.3.5 GARBAGE COLLECTION

The garbage collection process is responsible for deleting stale versions and advancing the FREE_SPACE_END pointer of the scatter table. Also, whenever an old version is deleted, the physical page that stores that old version is returned to the free page list available for allocating to transactions as work space. What constitute stale versions and how the garbage collection process operates on the scatter table are the subjects of the current discussion.

To determine old versions that are safe for deletion, the garbage collection process can make use of the time wall concept related to the Read-Only Protocol. Recall that to enable a read-only transaction to access any data segment without having to leave timestamps, be delayed or be aborted, a time wall, which is a set of times where each time is associated with a data segment, is computed and the read-only transaction is allowed to access data versions that are the most recent versions immediately prior to the times in the time wall. For example, to access data element d in a data segment D_i, and given that the time associated with D_i in the current time wall is $time_i$, the version of d which is the most recent version before $time_i$ is granted for access by the transaction.

It can be shown that the time wall, computed periodically, establishes a set of times that can be used to determine which older versions will never be accessed by active or future transactions, and therefore can be garbage collected. We will first re-state the following fact:

Fact: Given a time wall TW computed and released at $time_0$ by the procedure described in Section 6.2, and let TW = <$time_1$, $time_2$, ..., $time_n$>, where $time_i$ is the time component in TW associated with data segment D_i, then no update transactions rooted in the data segment D_i, for i = 1 to n, that started before $time_i$ are still active at time $time_0$.

The above statement confirms the fact that the version of a data element in data segment D_i that has a successor version which is prior to the time wall component $time_i$ in the currently applicalbe time wall will never be requested by any active and future transaction. Therefore this version is a stale version and can be garbage collected.

Armed with the above fact, the garbage collection process is a system process which is invoked whenver a new time wall is released. For every data element in a data segment D_i, the process traverses the version chain of that data element until a version is found which has a version timestamp prior to $time_i$ in the time wall. Then it deletes all versions of that data element prior to this version, if there is any.

To enable the garbage collection to be more efficient, the logical page id field of the scatter table entries in the Old Version Region of the the scatter table can be used to select the logical pages whose version chains are to be examined. The garbage collection process can look into only scatter table entries in the Old Versions Region and for each entry follow the version chain to see if there exists a successor version which is prior to the specified time value in the time wall. Given the fact that at any given time most data elements would probably have only one version in the database, selectively chosing version chains to be examined would greatly reduce the work of the garbage collection process by eliminating the need for it to look at all the data elements, but only those with more than one versions.

The Garbage Collection process would start at the entry pointed to by FREE_ENTRY_END + 1 in the Old Version Region of the scatter table and examine that entry according to the procedure described in the previous paragraph. Once it is determined that the entry can be garbage collected, the entry is marked free (e.g., setting a bit in the logical page id field) and FREE_ENTRY_END is incremented. This proceeds until either an entry is found that is not eligible for being garbage collected, or when FREE_ENTRY_END +1 is equated with FREE_ENTRY_BEGIN or it is already marked free. The Garbage Collection process can then be dormant till the next time wall is released.

Assuming that the operations of incrementing FREE_ENTRY_END and FREE_ENTRY_BEGIN are executed atomically, the garbage collection process will never interfere adversely with any other concurrent processes. While it is in no conflict at all with TIMESTAMPED_READ and NO_TRACE_READ, it can also be shown that it does not have to interfere with concurrent insertion operations by ATOMIC_COMMIT. We give the following facts to substantiate this statement:

(1) ATOMIC_COMMIT will never cause a version chain that the Garbage Collection process must follow to be distorted. This has been shown in discussing NO_TRACE_READ operations.

(2) Garbage_Collection will never garbage collect an entry that an ATOMIC_COMMIT process has just obtained from the free entry list (therefore in the domain of operation of the garbage collection

process) but has not written the new data in. This is because such entries would have been marked free.

7.3.6 SUMMARY OF THE MULTI-VERSION DATABASE IMPLEMENTATION

To summarize our implementation of the Multi-versin Database in the concurrency control facility, we emphasize that while ATOMIC_COMMIT and TIMESTAMPED_READ must be synchronized by the ENTRY_LOCK, NO_TRACE_READ and GARBAGE_COLLECTION do not have to use any lock or otherwise obey synchronization protocols, and therefore can always proceed without being blocked or incurring overhead for other processes.

7.4 IMPLEMENTING FUNCTIONS FOR MANAGING AND COMPUTING TIMESTAMPS

7.4.1 INTRODUCTION AND OVERVIEW

In this section we describe how timestamps are assigned or computed. Recall that in order to make use of Protocol H or Protocol L for accessing data elements outside of a transaction's own root data segment, it is necessary to compute an access timestamp which is different from the transaction's own initiation timestamp. Computation of these timestamps results in the need to maintain information on transaction history as well as pseudo-transaction constraints. Maintenance of these information is also the subject of the current section.

As shown in Figure 23 on page 141, four tasks (atomic work units) are related to timestamp management: INITIATION, COMMIT_TIMESTAMPING, COMPUTE_RTC and REQUEST_LTS. The first three share accesses to the Transaction Table, while the latter two share accesses to the Pseudo-transaction Table. Both tables are maintained strictly for the purpose of evaluating the A function to provide time ceilings used in the H protocol. We will therefore first describe in the following sub-section the algorithm of COMPUTE_RTC that evaluates the A function. The need for a fast evaluation algorithm leads to the requirement that the above information be presented to the COMPUTE_RTC module in a structure that is most suitable for the algorithm and the structure can be accessed by COMPUTE_RTC without being interfered by concurrent updates to the structure. A design that accomplishes this goal, and involves new tables compiled from the Transaction Table and the Pseudo-transaction Table for access by COMPUTE_RTC, is described in the final sub-section.

7.4.2 EVALUATING THE 'A' FUNCTION

When a transaction rooted in data segment D_i needs to access data elements in a higher data segment D_j, it uses the H protocol and accesses versions of these data elements in D_j that are immediately previous to a read time ceiling (RTC) computed by evaluating the function $A_{i,j}(TS)$, where TS is the initiation timestamp of the transaction. COMPUTE_RTC is a module which, when invoked, is given the source data

segment (e.g., D_i), the target data segment (e.g., D_j) and the transaction timestamp (e.g., TS), and will return the value of $A_i^j(TS)$.

There are two strategies concerning when to evaluate the A function. One is the static strategy, which computes at the initiation stage of a transaction all the read time ceilings the transaction will potentially use, and stores these values in the transaction header contained in the transaction's work space. The other is the dynamic strategy, which computes the read time ceiling only when a read access that requires it is issued by the transaction. Once computed, this time ceiling is then stored in the transaction header for possible future uses. Which strategy is to be preferred depends on the variability of the sets of higher data segments the transactions in the same class will access. However, regardless of the timing strategy, the basic function of the module COMPUTE_RTC remains the same.

The function $A_i^j(TS)$ is evaluated by the application of a series of the I_OLD function along the critical path from the source data segment D_i to the target data segment D_j. COMPUTE_RTC(i,j,TS) can be implemented recursively. At each step, the parent data segment, say D_h, of the current source data segment is found. An I_OLD function is invoked which finds the initiation time of the oldest active transaction at time TS in that parent data segment D_h. This initiation time is then passed as new TS to the next invocation of COMPUTE_RTC along with D_h as the new source data segment. The procedure COMPUTE_RTC therefore repeately

calls itself until the source data segment and the target data segment in the call coincide. The parent of a data segment along the critical path to the target data segment is obtained from a table PARENT, which is a static table derived from the data segment hierarchy.

An efficient implementation of COMPUTE_RTC depends on an efficient algorithm for finding the initiation time of the oldest active transaction at a given time. Due to difficulties involved in concurrent accesses to the Transaction Table and the Pseudo-transaction Table, two new tables, called the Summary Tables, are derived from them for use by COMPUTE_RTC. The data structures and operations on these tables, and how fast evaluation of I_OLD can be achieved, is presented in the following sub-section.

7.4.3 DATA STRUCTURES AND OPERATIONS

In this sub-section we describe the data structures of the four tables involved in timestamp managment and then discuss how concurrent accesses, updates and garbage collection are achieved.

The four tables are the Transaction Table (TT), the Transaction Summary Table (TT_SUM) which is derived from TT, the Pseudo-transaction Table (PT) and the Pseudo-transaction Summary Table (PT_SUM) which is derived from PT. Each data segment has a set of these tables associated with it. They are all implemented as ring buffers with associated

FREE_SPACE_BEGIN and FREE_SPACE_END pointers. Each entry in the tables is composed of two fields: Initiation time (IT) and Commit time (CT) corresponding to a particular transaction or psuedo transaction.

7.4.3.1 THE TRANSACTION TABLE (TT)

The Transaction Table is merely a running log of initiation times of recent transactions. For each incoming transaction, INITIATION provides an initiation timestamp and creates a new entry at the bottom of TT with the entry's commit time field set to a large value. Therefore TT is always ordered by the initiation time field. When a transaction is finished with its commit process, COMMIT_TIMESTAMPING provides a commit timestamp and locates the transaction's entry in TT and updates the commit time field. As shown in Figure 28, if the committing transaction is the oldest active transaction at the time of its commitment, then COMMIT_TIMESTAMPING removes the entries at the top of TT (by advancing the FREE_SPACE_END pointer) until the new top entry is the next oldest uncommitted transaction in the table. It also then creates appropriate entries in the Transaction Summary Table. Since the entry at the top of TT is always the current oldest active transaction, whether a committing transaction should be inserted into the Summary Table can be easily decided by the COMMIT_TIMESTAMPING process.

7.4.3.2 THE TRANSACTION SUMMARY TABLE (TT_SUM)

Original State of TT and TT_SUM

TT:

ITS	CTS
40	85
46	76
57	LOT
89	
90	
91	
109	
FREE AREA	
37	

TT_SUM:

ITS	CTS
25	79
37	
FREE AREA	

At time 119, Transaction with ITS = 37 is committed:

Step 1: Enter 119 into transaction's CTS field in TT

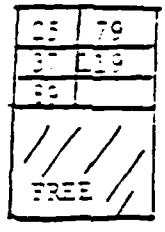
TT:

:	
37	119

(Continued from previous page)

Step 2: Since this transaction is at the top of the current table (i.e., its entry is right below FREE_SPACE_END,) therefore copy appropriate info into TT_SUM:

TT_SUM:



Step 3: Remove all entries until the next oldest tran. shows as the top entry in TT:

TT:

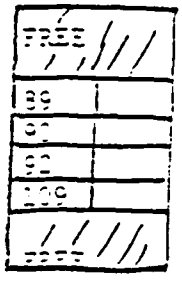


Figure 28. Deriving TT_SUM from TT.

The Transaction Summary Table contains the initiation times and the commit times of transactions that are 'time-dominant'. A time-dominant transaction is one which at some point in time has been the oldest active transaction in a data segment. Entries are inserted in TT_SUM when a time-dominant transaction commits. Since time-dominant transactions will commit in the order of their initiation times, the insertion procedure ensures that TT_SUM is ordered by both the initiation time and the commit time fields.

As discussed under The Transaction Table, insertion of entries into TT is performed by COMMIT_TIMESTAMPING. Every time COMMIT_TIMESTAMPING commits a transaction that is at the top of the Transaction Table, that transaction's commit timestamp is inserted and a new entry with the initiation time field set to be the initiation time of the next oldest active transaction is created.

7.4.3.3 PSEUDO-TRANSACTION TABLE (PT)

Whenever the module REQUEST_LTS is invoked and passed with a source data segment D_i, a target data segment D_j, and a timestamp TS, REQUEST_LTS will guess at the value of $B_i^j(TS)$ and return it. However, enforcement of the guessed value involves pseudo transactions to be created in the transaction classes rooted in beween D_i and D_j whose sole function is to influence timestamp management. Therefore REQUEST_LTS must insert the initiation time and the commit time of a pseudo trans-

action associated with a data segment in the Pseudo-transaction Table of that data segment for use by other timestamp management modules.

PT is to be ordered by initiation times. Since pseudo transactions may not arrive in that order, REQUEST_LTS may have to shift entries when inserting a new entry.

Entries in PT are removed by the INITIATION process. As shown in fig Figure 29, when a real transaction is initiated, the INITIATION process also looks at the top of the Pseudo-transaction Table and removes all entries in the latter that have initiation times smaller than the newly initiated transaction. (This can be done because pseudo transactions are always assigned an initiation timestamp later than the time when it is created.) These entries are then inserted into the Pseudo-transaction Summary Table to be described below.

7.4.3.4 PSEUDO-TRANSACTION SUMMARY TABLE (PT_SUM)

The Pseudo-transaction Summary Table contains the initiation and commit times of pseudo transactions that have become current enough for use by COMPUTE_RTC. An entry in the Pseudo-transaction Table becomes current when a real transaction with an initiation time greater than that of the former has been initiated. Therefore inserting entries into PT_SUM is performed by the INITIATION process.

Original State:

PT:

ITS	CTS
205	305
209	309
257	357
FREE AREA	
125	225
136	236
297	297

PT_SUM:

ITS	CTS
25	125
39	139
FREE AREA	

A transaction with ITS = 198 has just been initiated:

Step 1: Copy all entries in PT that started before 198 to PT_SUM

PT_SUM:

25	125
39	139
125	225
136	236
197	297
FREE AREA	

Step 2: Remove all copied entries from PT:

PT:

205	305
209	309
257	357
FREE AREA	
125	225
136	236
197	297

Figure 29. Deriving PT_SUM from PT.

Like the Transaction Summary Table, the semantics of the insertion process enables PT_SUM to be ordered by both the initiation times and the commit times. These two summary tables are therefore in a desired form to be presented to COMPUTE_RTC.

7.4.3.5 USING THE TIMESTAMP MANAGMENT TABLES

The conflict matrix among the four operations INITIATION, COMMIT_TIMESTAMPING, COMPUTE_RTC and REQUEST_RTS under the strategy of deriving summary tables from the Transaction and Pseudo-transaction tables is shown in Figure 30. Comparing this figure with Figure 23 on page 141, one can see that the nature of the conflict between COMPUTE_RTC and other atomic work units has been changed from a direct conflict on PT and TT to be that on PT_SUM and TT_SUM.

Given the two summary table, the initiation time of the oldest active transaction rooted in a data segment D_i at a given time TS can be found by looking into the summary tables corresponding to D_i according to the following algorithm:

(1) Idnetify the current bottom entry of the two tables.

(2) For each of the tables, examine each entry from the bottom of the table up until an entry e with a commit timestamp earlier than TS is found. Then obtain the initiation time value of the entry that follows e.

	INIT	CTS	RLTS	CRTC
INIT	TT, PT PT_SUM	TT	PT	PT_SUM
CTS		TT, TT_SUM		TT_SUM
RLTS			PT	
CRTC				

TT: Trans. Table
PT: Pseudo-trans Table
TT_SUM: Trans. table Summary
PT_SUM: Pseudo-trans. Summary

Figure 30. Revised Conflict Matrix for Timestamp Managment Processes.

(3) Compute the minimum of the following three values: TS and the two initiation values obtained from the two tables. Return this minimum.

It can be shown that this algorithm does not interfere with concurrent insertion and garbage collection processes on the two summary tables. To prove this, we provide the following facts (Note that those properties of the summary tables that concern the following facts are missing from the original PT and TT and therefore non-interfering reading of the tables could not be achieved were COMPUTE_RTC to directly access PT and TT):

(1) Insertion to the summary tables always occur at the bottom of the table. Those newly inserted entries, however, will not be of interest to any concurrent COMPUTE_RTC process because the latter must be searching for an entry earlier than the ones being created.

(2) Garbage Collection of the summary tables always occur from the top of the tables. However, it will not interfere with the concurrent COMPUTE_RTC processes because Garbage Collection only strip off those stale entries (i.e., entries corresponding to transactions committed before certain timewall) that would no longer be the target entries being searched for by a concurrent COMPUTE_RTC.

To conclude, we have, by designing data structures that are best suited for COMPUTE_RTC, enables the process of evaluating A function to proceed without having to be interfered by any concurrent activity.

8.0 HIERARCHICAL DATABASE DECOMPOSITION METHODOLOGY

8.1 INTRODUCTION AND SUMMARY OF RESULTS

As concluded in previous chapters, successful application of the HDD approach to concurrency control depends on an intelligent choice of the data segment hierarchy on which the protocols are based. How this choice can be arrived at is the subject of the current chapter on decomposition methodology. In essence, we would like to develop an algorithm that, given the description of a database application which is captured in a specified form, will efficiently compute a favorable database segmentation and a corresponding data segment hierarchy for use by the HDD concurrency control technique.

This chapter is composed of four sections. The next section develops a formal model for describing the problem of hierarchical decomposition. The model is an integer programming model and takes the description of a database application in terms of 'data components' and 'transaction types'. The objective of the integer programming is to find an optimal scheme for clustering these data components into a hierarchy of data segments. The complexity of the model is analyzed in the subsequent section, which concludes that the decomposition problem is NP-hard, therefore optimization is practical only for very small problems. To

remedy this situation, a heuristic procedure is developed in section four which attempts to find a 'good' solution in a reasonable amount of time.

8.2 A FORMAL MODEL OF THE HIERARCHICAL DATABASE DECOMPOSITION PROBLEM

In this section, the problem of finding a favorable database segment hierarchy given the description of a target database application is formulated into an optimization problem.

The formulation assumes that an analysis of the database application has been performed and the content of the database as well as its accesses by transactions is understood and documented. This document must describe three aspects of the application:

(1) Data components, which are mutually exclusive and collectively exhaustive subsets of the data items in the database where data items within the same subset exhibit generic similarities. For example, an EMPLOYEE relation could be considered a data component. Data components will constitute the smallest units for consideration as data segments (i.e., data segments are clusters of data components.)

(2) Transaction types, which are also mutually exclusive and collectively exhaustive subsets of the update transactions to be run in the database. Transactions within the same transaction type

have similar database access requirements. Transaction types will constitute the smallest units for consideration as transaction classes.

(3) Access frequencies, which describe the frequencies of read and write accesses from each of the transaction types to each of the data components.

A database application where the use of the HDD concurrency control technique potentially produces gains is one in which accesses from some transaction types to some data components are read-only. These read-only accesses are the source of the usage of the H protocol. However, given that the database application exhibits the potential for applying the HDD concurrency control technique, one must also discover how data components ought to be merged and fit into a hierarchy so as to maximize the use of the H protocol and minimize the use of the L protocol.

This problem of discovering the optimal data segment hierarchy can be formulated as an integer programming problem in which the frequency of usage of the H protocol and that of the L protocol can be expressed given an assignment of the data components to a data segment hierarchy. Formally, given a database application which is composed of a set of data components DC_1, DC_2, ..., DC_n, and a set of transaction types TP_1, TP_2, ..., TP_m, and given that the read and write access frequencies of transaction type TP_i to data component DC_j are known as $r_{i,j}$ and $w_{i,j}$,

we want to find an assignment of the n data components to a data segment hierarchy DSH consisting of up to n data segments DS_1, DS_2, ..., DS_n, where it is assumed that DS_{i1} is higher than DS_{i2} in DSH if i1 < i2, (i.e., we assume that the index of a data segment is an indication of the position of that data segment in the hierarchy,) such that the access gain function, defined as a weighted sum of the frequencies of the usage of the H protocol and the L protocol, is maximized. In other words, let

$r_{i,j}$ = frequency of read accesses from TP_i to DC_j, i = 1,...,m, j = 1,...,n,

$w_{i,j}$ = frequency of write accesses from TP_i to DC_j, i = 1,...,m, j = 1,...,n.

The problem is to solve for the following set of decision variables X and Y:

$$X_{j,k} = \begin{cases} 1 \text{ if } DC_j \text{ is assigned to } DS_k \\ 0 \text{ otherwise} \end{cases}$$

for j = 1,...,n and k = 1,...,n, and

$$Y_{i,k} = \begin{cases} 1 \text{ if } TP_i \text{ is assigned to } DS_k \\ 0 \text{ otherwise} \end{cases}$$

for i = 1,...,m and k = 1,..., n.

Subject to the following constraints:

(1) Each data component must be assigned to one and only one data segment,

$$\text{i.e., } \sum_{k=1}^{n} X_{j,k} = 1 \text{ for } j = 1,\ldots,n.$$

(2) Each transaction type must be rooted in one and only one data segment,

$$\text{i.e., } \sum_{k=1}^{n} Y_{i,k} = 1 \text{ for } i = 1,\ldots,m.$$

(3) A transaction type will not write into a data segment higher than its own root segment,

$$\text{i.e., } w_{i,j} * X_{j,k} * \sum_{h=k+1}^{n} Y_{i,h} = 0 \text{ for all } i,j,k.$$

The objective is to maximize the number of read accesses to higher data segments from transactions rooted in lower data segments (i.e., usage of the H protocol) and to minimize the number of read and write accesses to lower data segments from transactions rooted in higher data segments (i.e., usage of the L protocol.) With the above definitions, the frequency of use of the H and the L protocols can be expressed as follows:

(1) Frequency of the H protocols

$$= \sum_{i=1}^{m} \sum_{j=1}^{n} \sum_{k=1}^{n} \sum_{h=k+1}^{n} r_{i,j} * X_{j,k} * Y_{i,h}$$

(2) Frequency of the L protocols

$$= \sum_{i=1}^{m} \sum_{j=1}^{n} \sum_{k=1}^{n} \sum_{h=1}^{k-1} (r_{i,j} + w_{i,j}) * X_{j,k} * Y_{i,h}$$

We will formulate the objective function as the first term above minus the second term weighted by the 'distance' in the segment hierarchy between the root data segment and the data segment being accessed, and by p, the cost ratio of the L protocol over the H protocol. (i.e., The gains of the use of one H protocol will be offset by the cost of the use of p times the L protocols of 'distance' one.) Formally, let the weight of using the L protocol to access a lower data segment DS_{j2} by a transaction rooted in a higher data segment DS_{j1} be $d_{j1,j2}$. Then $d_{j1,j2} = p(j1-j2)$ for $j1 < j2$. The objective function can be formulated as follows:

$$\text{MAX} \sum_{i=1}^{m} \sum_{j=1}^{n} \sum_{k=1}^{n} \left(\sum_{h=k+1}^{n} r_{i,j} * X_{j,k} * Y_{i,h} - \sum_{h=1}^{k-1} (r_{i,j} + w_{i,j}) * X_{j,k} * Y_{i,h} * d_{h,k} \right)$$

Now we summarize the above formulation by presenting the entire problem formulation for the HAO problem as follows:

$$\text{MAX} \sum_{i=1}^{m} \sum_{j=1}^{n} \sum_{k=1}^{n} \left(\sum_{h=k+1}^{n} r_{i,j} * X_{j,k} * Y_{i,h} - \sum_{h=1}^{k-1} (r_{i,j} + w_{i,j}) * X_{j,k} * Y_{i,h} * d_{h,k} \right)$$

Subject to

$$(1)\ \sum_{j=1}^{n} X_{j,k} = 1, \quad X_{j,k} = 0,1$$

$$(2)\ \sum_{i=1}^{m} Y_{i,k} = 1, \quad Y_{i,k} = 0,1$$

$$(3)\ w_{i,j} * X_{j,k} * \sum_{h=k+1}^{n} Y_{i,h} = 0$$

for all i = 1,...,m, j = 1,...,n, and k = 1,...,n.

The above formulation will from now on be referred to as the Hierarchical Assignment Optimization Problem, or the HAO problem.

8.3 COMPLEXITY OF THE HIERARCHICAL ASSIGNMENT OPTIMIZATION PROBLEM

The integer programming optimization problem presented in the previous section, once solved given a particular set of r and w parameters, will produce the optimal segment hierarchy for use by the HDD concurrency control technique. However, the complexity of the HAO problem appears to prohibit an exhaustive search for an optimal solution when the problem is of a non-trivial size. In this section, the complexity of the HAO problem is analyzed. The theory of NP-completeness is applied to show that the HAO problem is NP-equivalent, which amounts to asserting that a polynomial time algorithm exists for this problem if and only if such an algorithm exists for the hardest problems in the

class NP. This conclusion signals the need for an efficient heuristic procedure for producing 'good' solutions for the HAO problem.

We will first review and clarify the concept of NP-hardness. A problem *p* is said to be NP-hard (i.e., at least as hard as an NP-complete problem) if there exists an NP-complete problem *p'* such that *p'* is Turing reducible to *p* <Garey79>. Turing reducibility from *p'* to *p*is established by asserting the existence of a (polynomial time) Turing reduction algorithm. A Turing reduction algorithm from a search problem *p'* to another search problem *p* is an algorithm A that solves *p'* by using a hypothetical subroutine S for solving *p* such that if S were a polynomial time algorithm for *p*, then A would be a polynomial time algorithm for *p'*. Intuitively, this means that one can show that a problem *p'* is Turing reducible to *p* by demonstrating that *p'* can be solved by calling a polynomial number of times the algorithm that solves *p* as a subroutine.

Based on the above discussion, to prove that HAO is NP-hard, one must find an NP-complete problem *p'* and show that *p'* is Turing reducible to HAO. To facilitate the proof procedure, we will re-phrase the integer programming formulation of the HAO problem. The new formulation will express the sets of 0-1 decision variables X and Y as functions f and g where f assigns data components to data segments and g maps transaction types to their root data segments. In addition, the set of weight val-

ues d is augmented in order to simplify the expresssion of the objective function. This new formulation is given below:

The Hierarchical Assignment Optimization (HAO) Problem:

Instance: Given $r_{i,j}$, $w_{i,j}$, $i = 1,..,m$, $j = 1,...,n$, and $d_{h,k}$, $h,k = 1,...,n$, where

$$d_{h,k} = \begin{cases} 0 & \text{if } h = k \\ 1 & \text{if } h > k \\ p(h-k) & \text{if } h < k \end{cases}$$

where p is a positive number.

Objective: Find functions f and g where

$f: \{1,2,...,n\} \rightarrow \{1,2,...,n\}$, and

$g: \{1,2,...,m\} \rightarrow \{1,2,...,n\}$, such that

(HAO-C1) for all i and j where $w_{i,j} \neq 0$, $g(i) \leq f(j)$, and

(HAO-Obj) the weighted sum $\sum_i \sum_j (r_{i,j} + w_{i,j}) * d_{g(i),f(j)}$ is maximized.

Theorem. The Hierarchical Assignment Optimization problem is NP-hard.

Proof. The proof consists of two parts. The first part proves that the decision problem counterpart of the HAO problem, called the Hierarchical Assignmemt problem, or the HA problem, is NP-complete. The second part, using relatively standard arguments, proves that the HA problem is Turing reducible to the HAO problem, therefore by the definition of NP-hardness, HAO is NP-hard.

To prove the first part, we will first formulate the decision problem counterpart of the HAO problem, namely, the Hierarchical Assignment Problem as follows:

The Hierarchical Assignment (HA) Problem:

Instance: Given $r_{i,j}$, $w_{i,j}$, i = 1,..,m, j = 1,...,n, a positive intenger Q, and $d_{h,k}$, h,k = 1,...,n, where

$$d_{h,k} = \begin{cases} 0 & \text{if } h = k \\ 1 & \text{if } h > k \\ p(h-k) & \text{if } h < k \end{cases}$$

where p is a positive number.

Question: Does there exist two functions f and g where

f: {1,2,...,n} -> {1,2,...,n}, and

g: {1,2,...,m} -> {1,2,...,n}, such that

(HA-C1) for all i and j where $w_{i,j} \neq 0$, $g(i) \leq f(j)$, and

(HA-C2) the weighted sum $\sum_i \sum_j (r_{i,j} + w_{i,j}) * d_{g(i),f(j)}$ is at least Q?

Lemma. HA is NP-complete.

Proof. (i) NP class membership: It is obvious that given f and g, the decision problem can be verified in polynomial time by merely (a) computing the sum and checking it against Q, and (b) verify that $g(i) \leq f(j)$ for all i and j where $w_{i,j} \neq 0$.

(2) NP hardness: HA is NP-hard if one can find a problem *p* known to be NP-complete such that *p* is (polynomial) transformable to HA. We will present a proof using the technique of restriction, which identifies a restricted instance of HA and equate it to a known NP-complete problem, namely, the Maximum Cut (MCUT) problem.

Restricting the HA problem:

Step 1: Let the parameters $w_{i,j}$ be restricted such that there exists a function ROOT: {1,2,...,m} -> {1,2,...n} such that $w_{i,j} = 0$ for all j ≠ ROOT(i). (i.e., Every transaction type TP_i writes in only one data component, namely, ROOT(i).)

The purpose of this restriction is to enable the dropping of the function g and the constraint HA-C1 from the HA problem. We achieve this by restricting the problem so as to make function g completely derivable from the function f given another known function ROOT. The reason that we could do this is as follows. Since $d_{h,k}$ is positive only when h > k, there is an incentive built into the problem, due to the nature of the objective function, to maximize g(i) so as to increase the chance that the multiplier $d_{g(i),f(j)}$ actually used in computing the objective function is positive. However, maximizing g(i) is subject to constraint (HA-C1), i.e., g(i) cannot be greater than f(j) for all j where $w_{i,j} \neq 0$. Combining this observation with the current restriction that $w_{i,j} = 0$ for all j ≠ ROOT(i), it is clear that the optimal choice

of g(i) under the current restriction is g(i) = f(ROOT(i)). Therefore, in this restricted version of the HA problem, the function g is entirely derivable from function f, and the constraint HA-C1 can be dropped.

Step 2: We will subject the problem to the following additional two restrictions:

m = 2

n > m.

With these restrictions, and the fact that max(f(j)) ≤ min(m,n), the function f can now be defined as follows:

f: {1,2,...,n} -> {1,2}.

The reason for f(j) to be bounded from above by min(m,n) is that the number of data segments contained in an optimal data segment hierarchy cannot be more than the number of transaction types, because if this were not the case, then there would exist some data segment not rooted by any transaction type that can be merged with its parent data segment in the segment heirarchy without incurring a penalty to the objective function, and thereby reducing the number of data segments in the optimal data segment hierarchy.

The purpose of this step 2 restriction is to transform the assignment problem into a 2-partition problem.

Step 3: Let $q_{i,j} = r_{i,j} + w_{i,j}$. The last set of restrictions to be applied is as follows:

$p < 1$, and

$q_{i,j} = q_{j,i}$ for all i and j.

These retrictions are applied to transform the problem from a directed partition problem into a simple (i.e., undirected) one.

The original problem with these three steps of restrictions can now be reformulated as follows:

The Restricted HA problem:

Instance: Given $r_{i,j}$, $w_{i,j}$, where i = 1,2 and j = 1,2,...,n, and n > 2, a function ROOT: {1,2} -> {1,2,..., n}, a set of d values where $d_{1,1} = d_{2,2} = 0$, $d_{2,1} = 1$ and $d_{1,2} = p$ where $0 < p < 1$, and a positive integer Q, and that these parameters satisfy the constraints (1) $r_{i,j} + w_{i,j} = r_{j,i} + w_{j,i}$ for all i,j and (2) $w_{i,j} = 0$ for all pairs of i and j where $j \neq ROOT(i)$.

Question: Does there exist a function f: {1,2,...,n} ->{1,2}

such that $\sum_i \sum_j (r_{i,j} + w_{i,j}) * d_{f(root(i)),f(j)}$ is at least Q?

We will now show that the Restricted HA problem is identical to the Maximum Cut problem defined below:

The Max Cut (MCUT) problem:

Instance: Given a graph G = (V,E), where V = {1,2,..,v} , weights on edges w(k,j) ϵ positive integers for all edges e(k,j) ϵ E, and a positive integer Q'.

Question: Is there a partition of V into disjoint sets V1 and V2 such that the sum of the weights of the edges from E that have one end point in V1 and one end point in V2 is at least Q'?

Let p' be the smallest integer such that p'/(1-p) is an integer. The process of equating MCUT To Restricted-HA is as follows:

(1) Let V = {1,2,..., n}.

(2) Let $w(k,j) = p' * (r_{i,j} + w_{i,j})$, where k = ROOT(i).

(3) Let Q' = Q * p'/(1-p).

Since MCUT is NP-complete <Garey79> and Restricted-HA is identical to MCUT, we conclude that MCUT is transformable to HA, and therefore HA is NP-complete. *Q.E.D.*

Now we must show that HA is Turing reducible to HAO to complete the proof of our theorem. This, however, is obvious because if there existed a polynomial algorithm A that solves HAO, then HA can be easily answered by calling A once and compare the optimal value produced by A with the decision threshould Q in HA. *Q.E.D.*

8.4 A HEURISTIC ALGORITHM FOR THE HIERARCHICAL ASSIGNMENT OPTMIZATION PROBLEM

In this section we will present a heuristic procedure for producing a good but not necessarily optimal solution for the hierarchical assignment optimization (HAO) problem introduced in the previous sections. The heuristic procedure is based on an analysis of the special structure of the HAO problem. This analysis sheds light on two aspects of the problem: first, how one may find a potentially good initial solution for the problem; second, how one may improve on this initial solution.

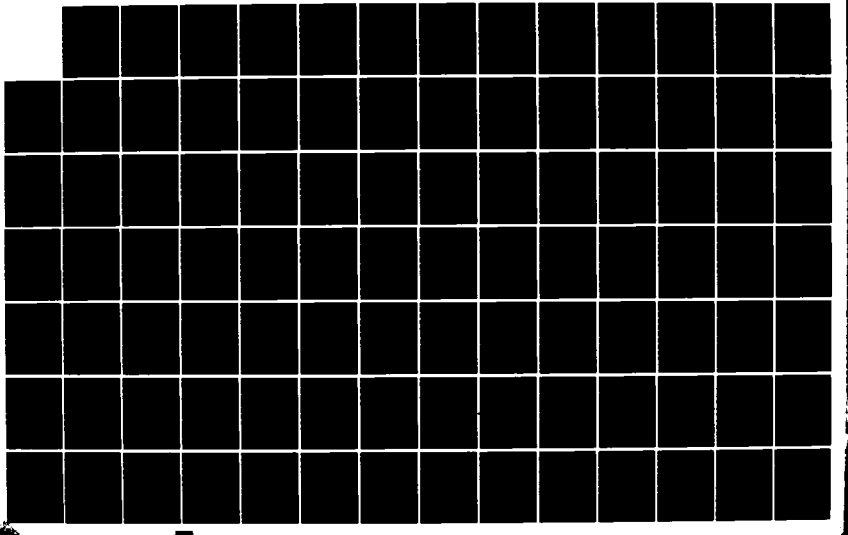
8.4.1 THE TABULAR REPRESENTATION OF A SOLUTION TO THE HAO PROBLEM

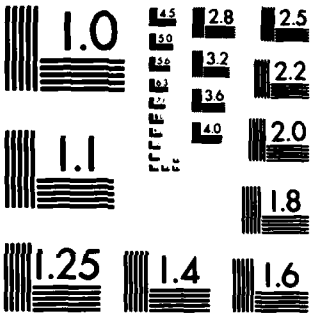
We will show that any solution to the HAO problem can be represented in a tabular form in which the individual elements in the two summation components of the objective function are separately displayed. Based on this observation, the objective of the heuristic procedure is to identify a solution whose tabular form possesses the property that the region representing the positive component of the objective function is maximized while the region representing the negative component is minimized. This observation is used to guide the design of our heuristic procedure. We will therefore first briefly describe this tabular form of solution to the HAO problem.

Let the parameters of a database application be represented by an access frequency table in which there are m rows reprenting m transaction types and n columns representing n data components. Each cell of

AD-A150 612 THE HIERARCHICAL DATABASE DECOMPOSITION APPROACH TO DATABASE CONCURRENCY..(U) ALFRED P SLOAN SCHOOL OF MANAGEMENT CAMBRIDGE MA CENTER FOR I.. M HSU DEC 84 3/4

UNCLASSIFIED CISR-TR-16 N00039-83-C-0463 F/G 9/2 NL





Three data components: DATA, LEAF, NON_LEAF

Three transaction types: KEY_UPD, NON_KEY_UPD, TREE_UPD

Access Frequency Table:

DC / TP	DATA	LEAF	NON_LEAF
KEY_UPD	(30,30)	(30,30)	(120,0)
NON_KEY_ UPD	(70,70)	(140,0)	(280,0)
TREE_UPD	(0,0)	(20,0)	(20,20)

(a,b): (frequency of read, frequency of write)

Figure 31. An example of an Access Frequency Table.

the table is composed of two values $r_{i,j}$ and $w_{i,j}$. An example of such a table is shown in Figure 31.

Given any feasible solution $S = \{f, g\}$ to an HAO problem with an access frequency table T. where f and g are defined as in the previous section, we can partition the cells in T into three kinds:

(1) The E cells: A cell (i,j) in T is an E-protocol cell, (or E cell) if there exists k such that f(i) = k and g(j) = k.

(2) The H cells: A cell (i,j) in T is an H-protocol cell, (or H cell) if g(i) = k2 and f(j) = k1 and k1 < k2.

(3) The L cells: A cell (i,j) in T is an L-protocol cell (or L cell) if g(i) = k1 and f(j) = k2 and k1 < k2.

In essence, a cell is an E cell (H cell, L cell) if, under solution S, the accesses to the database described by that cell have to use the E protocol (the H protocol, the L protocol.) By the definition of a feasible solution, all $w_{i,j}$ where (i,j) is an H cell must be zero. The objective function value corresponding to a feasible solution S can be given as the following:

$$(\text{Obj}) \quad \sum_{\substack{(i,j) \text{ is} \\ \text{an H cell}}} r_{i,j} - \sum_{\substack{(i,j) \text{ is} \\ \text{an L cell}}} (r_{i,j} + w_{i,j}) * d_{g(i),f(j)}$$

The tabular representation of a solution S for a problem with an access frequency table T is a permutation of T in which these three kinds of cells are neatly bundled together. Formally, the tabular representation

of an access table T given a solution S, denoted as $p(T,S)$, is a table produced by permutating rows and columns in T using the following rules:

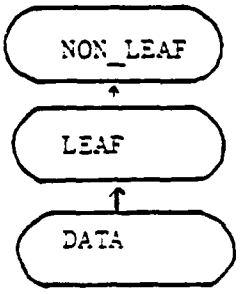
(1) Permutate columns of table T such that if $f(j1) = k1$ and $f(j2) = k2$ and $k1 < k2$ then column j1 is before column j2 in the representation.

(2) Permutate the rows of table T such that if $g(i1) = k1$ and $g(i2) = k2$ and $k1 < k2$ then row i1 is before row i2 in the representation.

It is interesting to see that in the tabular representation the E cells partition the rest of the cells diagonally into two regions, where the lower-left region corresponds to the collection of H cells, and the upper-right region corresponds to the collection of L cells. An example of the tabular representation of a solution to the example table shown in Figure 31 is shown in Figure 32.

The property that the three kinds of cells are bundled into regions in the tabular representation enables a visualization of a solution S. Solving for the HAO problem with a table T can now be considered as a process of shuffling the columns and rows in T and creating a diagonal region (1) below which the constraint on H cells is enforced and (2) that the values in the cells in the lower-left region is maximized w.r.t. those in the upper-right region. The solution captured in the tabular representation then can be read from the permutated sequence of the columns and the span of the diagonal region. This pictorial expla-

Assuming a solution to the above problem is the following data segment hierarchy DSH* :



The tabular solution with each regions delineated is as follows:

DC TP	NON_LEAF	LEAF	DATA
TREE_UPD	(20,20)	(20,0)	(0,0)
KEY_UPD	(120,0)	(30,30)	(30,30)
NON_KEY_UPD	(280,0)	(140,0)	(70,70)

Upper right region, or the L region

Diagonal region, or the E-cell region (shaded area)

Lower left region, or the H-cell region

Figure 32. An example of the tabular representation of a solution.

nation of the solution leads to the devise of the following heuristic procedure.

8.4.2 AN OVERVIEW OF THE HEURISTIC PROCEDURE

The heuristic procedure consists of two stages as shown in Figure 33. The first stage attempts to find a 'favorable' permutation of the data components. Once this permutation # is found, where #(i) denotes the index of the data component chosen to be the i-th data component in the permutation, the function f is simply one which assigns data components to data segments in a one-to-one manner where data component $DC_{\#(i)}$ is assigned to data segment DS_i. The function g, derivable from function f, assigns the highest data segment in f that a transaction type writes in to be its root data segment. No merging of multiple data components to form one data segment is considered at this stage. Therefore the solution produced by this stage would result in a data hierarchy consisting of n data segments, and a transaction type is assigned to be rooted in the highest data segment it has to write into.

The second stage is one in which a local interchange/merging process of a certain 'degree' K is performed on the first-stage solution $S^{(1)}$ to improve on the latter and to arrive at a final solution $S^{(2)}$. If the objective function value of the solution $S^{(2)}$, denoted as $v(S^{(2)})$, is non-positive, then the heuristic procedure has failed to produce a feasible data segment hierarchy that will produce net gains.

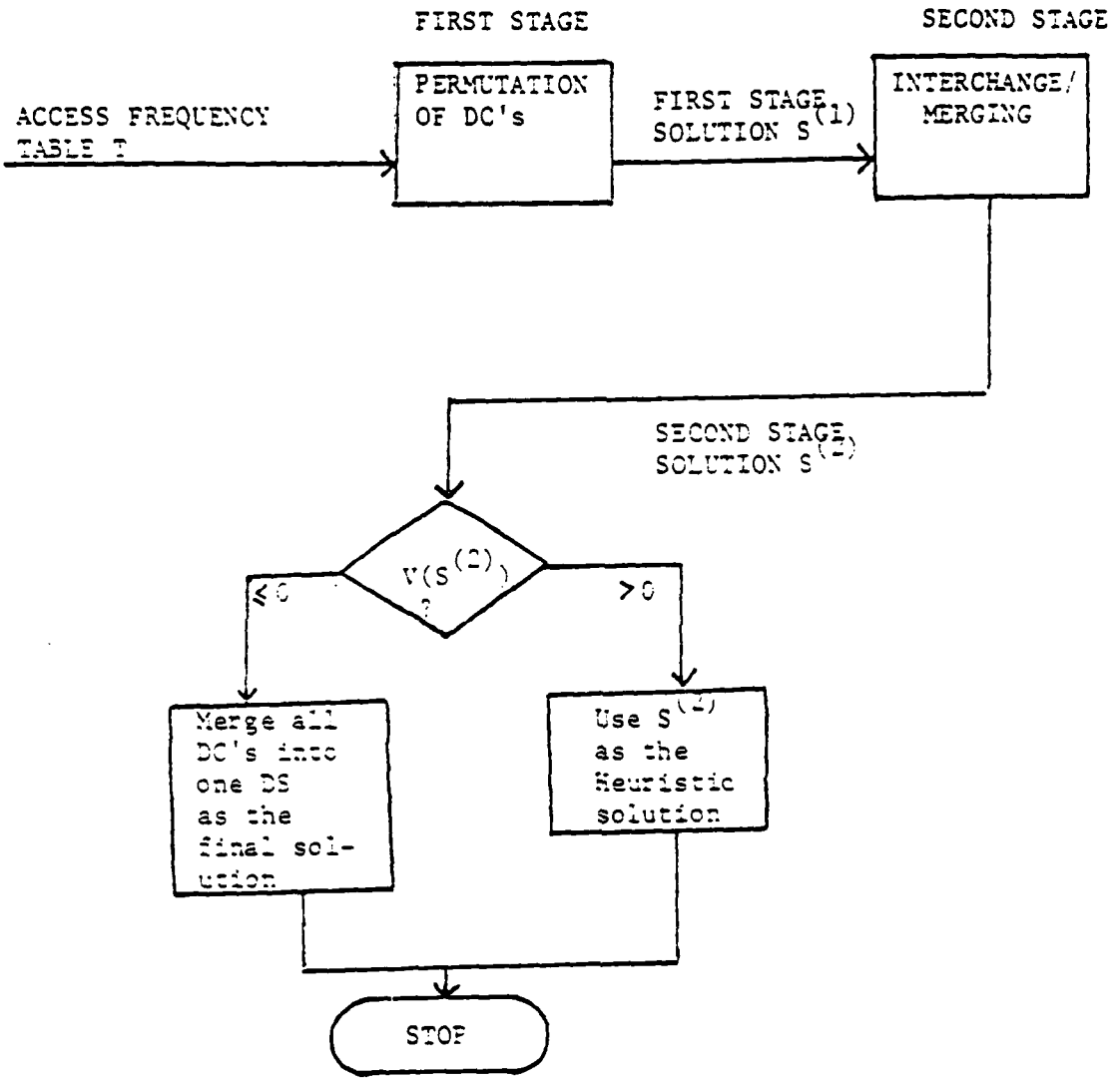


Figure 33. A two-stage heuristic procedure for solving HAC problem.

Note that $v(S^*)$, where S^* is the optimal solution to the HAO problem, is always bounded below by 0. This is because a solution in which all data components are merged into one data segment is always a feasible solution which produces an objective function value of zero. Therefore if $v(S^{(2)}) < 0$, then one can always discard $S^{(2)}$ and set the solution to the problem to be a complete merge of all data components into one data segment.

8.4.3 THE FIRST STAGE OF THE HEURISTIC PROCEDURE

The objective of the first stage is to find a reasonable permutation # of the data components. The 'reasonableness' of this initial solution hinges upon the likelihood of the existance of a 'good' solution $S^{(2)}$ in which those data components that are close to each other in the permutation # are either merged or assigned to data segments that are close to each other. If such a goold solution does exist, then it can be found in the second stage from the first-stage solution $S^{(1)}$ with a proper choice of the degree k.

In our algorithm, # is chosen in a sequential manner by selecting, based on some rules, data component j1 to be #(1), then data component j2 to be #(2), and so on. Therefore this process is composed of a total of n steps, at the i-th step of which #(i) will be selected from the remaining pool of data components. At any step we must evaluate the remaining data components to find the one that might produce the biggest

gain when assigned to #(i). To avoid combinatorial explosion, we wish to limit our consideration to just the immediate impact of the next selection on the value of the objective function, without regard to the impact of the ordering in # of all the yet-unselected ones. We will compute this immediate impact by pretending that the remaining data components would be merged once the next selection is made, and therefore the component selected should be the one that will make the biggest contribution to the objective function under this assumption. This procedure embodies the philosophy of a 'greedy' move often encountered in the design of heuristic procedures, which selects the next node to be evaluated in the branching tree by computing the immediate gains only.

An additional rationale for this algorithm is that it is recognized that the optimal solution to the HAO problem is one in which those data components that many transaction types have high-volume read-only accesses to are assigned to data segments that are relatively higher than some others, so as to enable these read-only accesses to use the H protocol and therefore increase the value of the objective function. The algorithm in stage 1 has the property that it tends to select data components that many transaction types generate high-volume read-only accesses to as first ones in the sequence #. This increases the chance that the initial solution is close to some good solution.

This procedure is formally described as follows. Let $S^{(1,i)}$, $i \geq 1$, be the solution produced right after the i-th component of # is computed, i.e.,

$$S^{(1,i)} = \{ f^{(1,i)}, g^{(1,i)} \mid f^{(1,i)}(\#(j)) = j \text{ for } j \leq i,$$

$$f^{(1,i)}(j) = i+1 \text{ for all } j \text{ not in } \{\#(1), \#(2),..,\#(i)\},$$

$$g^{(1,i)}(k) = \min \{f^{(1,i)}(j) \mid w_{k,j} \neq 0\}\}$$

The heuristic rule for choosing #(i) is as follows: choose #(i), $1 \leq \#(i) \leq n$, #(i) not in {#(1), #(2), ..., #(i-1)}, such that $v(S^{(1,i)})$ is maximized.

8.4.4 THE SECOND STAGE OF THE HEURISTIC PROCEDURE

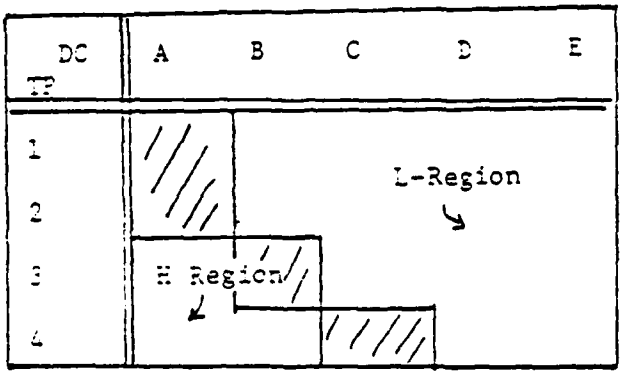
In the second stage, attempts are made to improve the solution produced by the first stage through performing up to n local interchange/merging steps. Each of these steps is performed on top of the intermediate solution produced in the previous step and is guaranteed to generate a solution at least as good as the previous one. An informal description of the algorithm is as follows.

At the first step, the solution $S^{(1)}$ produced by the first stage is taken as the initial solution. The bottom K data segments in this solution, for a fixed and small number K, are examined and evaluations are made for all possible re-arrangements, including merging of these K data segments, while keeping the rest of the data segments where they were in

the inital solution. An arrangment of these K data segments is then chosen which is concactenated from the bottom with the rest of the data segments in the initial solution to form the solution for the first step. At the next step, taking the solution produced by the previous step as the initial solution, the same process is repeated for the K data segments that are as close to the bottom as possible but subject to the constraint that at least one new data segment that was outside of the scope of the K data segments considered in the previous step is involved. This process is repeated until all data segments in the initial solution $S^{(1)}$ has been included in at least one of the local interchange/merging steps.

The rationale behind this algorithm is the belief that it is very likely that the initial solution produced in the first stage is a rough cut of a good solution that local refinement of the sort described in the above algorithm can bring out. Referring to the tabular representation of the solution, each step in this second stage looks at a bundle of data segments and evaluates the result of different ways of rearranging these data segments. Then in the next step, moves the bundle to be considered up by one data segment. Every time a bundle is examined, the area covered by that bundle is refined to make the left-lower region in the tabular representation even 'heavier', thereby bringing the solution closer to a desired form. An example of this process in its tabular form is shown in Figure 34.

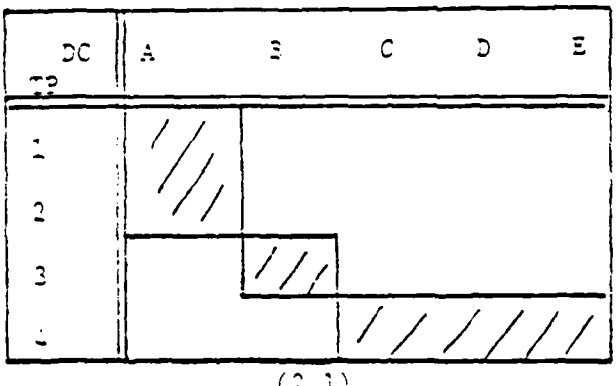
$S^{(1)}$ in its tabular representation:



$S^{(1)}$ =

Let K = 3.

First Step; Examine all possible rearrangement of the lower 3 data segments in solution $S^{(1)}$. The arrangement that merges segments C, D, and E together is chosen, and the tabular rep. of $S^{(2,1)}$ is:



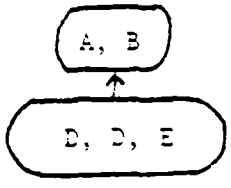
$S^{(2,1)}$ =

(Note that in $S^{(2,1)}$ the L region is smaller)

(Continued from previous page)

Second Step: Examine all possible rearrangements of the 3 data segments in $S^{(2,1)}$, and choose a new arrangment where A and B are merged. i.e.,

$S^{(2,2)}$ =



The tabular rep. of $S^{(2,2)}$ is:

DC / TP	A	B	C	D	E
1			L Region		
2					
3					
4	H Region				

Since all data segments in $S^{(1)}$ have been included in the scope of consideration in at least one step in the second stage, the process stops and $S^{(2)} = S^{(2,2)}$.

Figure 34. An Example of the Second stage of the Heuristic Procedure.

The algorithm is formally described as follows: Let the solution produced by step i of the second stage be $S^{(2,i)}$, the number of data segments in $S^{(2,i)}$ be denoted as $q^{(i)}$. At the beginning of step i, given $S^{(2,i-1)}$, the task is to find a function $G^{(i)} = \{ Q, Q+1, \ldots, Q+K-1 \} \rightarrow \{ Q, Q+1, \ldots, Q+K-1 \}$ where $Q = \min \{ n-K-i+2, q^{(i-1)}-K+1 \}$, such that $v(S^{(2,i)})$ is maximized, where $S^{(2,i)} = \{f^{(2,i)}, g^{(2,i)}\}$ and $f^{(2,i)}$ is defined as follows:

$$f^{(2,i)}(j) = \begin{cases} f^{(2,i-1)} & \text{for } j < Q \\ G^{(i)}(j) & \text{for } Q \le j \le Q+K-1 \\ f^{(2,i-1)} - e & \text{for } j > Q+K-1, \end{cases}$$

where $e = (Q+K-1) - \text{Max} \{ G^{(i)}(j) \}$

The complexity of the second stage algorithm depends on the choice of K, as the total number of evaluation to be performed in this stage is in the order of $n * K! * (C(k,1) + C(k,2) + \ldots + C(K,K))$, clearly non-polynomial in K. However, the larger K is, the higher is the probability that the final solution found at the end of the second stage is closer to the optimal solution. Therefore choice of K represents a tradeoff between closeness to the optimal solution and the consideration for computing cost.

9.0 APPLICATIONS OF THE HDD APPROACH TO DATABASE CONCURRENCY CONTROL

This chapter is devoted to application studies. The purpose of these studies is to illustrate how the HDD approach can be effectively applied. In the first of these three cases, we show how the hierarchical algorithm can be used to improve performance of a database with a heavily congested index area. The case is an application of the hierarchical method in a simple cyclic database partition composed of essentially two data segments. In the second case, we show how the HDD approach can be used to restructure a banking application from its current segregated state of operation into a real-time database operation without increasing data contention. It also exemplifies the advantage of using the HDD approach to structure databases. In the third case, we show how database computers with large-scale multiprocessing activities can benefit from a design guieded by the concept of hierarchical decomposition - how the latter helps reduce the inter-processor interference due to database concurrency control.

9.1 CONCURRENCY CONTROL IN AN INDEXED DATABASE

In this section we present an application of the hierarchical decomposition approach to concurrency control in a database management system

that uses multi-level indices as access paths. Dividing the database into an index data segment and a data record segment, we obtain a simple decomposition of the database into two data segments. This case therefore illustrates an application of the HTS algorithm to a simple two-data segment cyclic data partition. The motivation behind choosing this arrangement as one of the cases for demonstration is two-fold. First, the indexed database is a structure that is popular in database management systems. Second, this case is easy to understand and is simple in terms of the structure of the data segment hierarchy (consisting of only two data segments) but clearly demonstrates one particular aspect of the effectiveness of the HTS algorithm - how it can be used to relieve contention in a high-traffic area of the database, in this case, the index area.

The structure of this section is as follows. We will in the first subsection review the basic data structures and operations on indices organized as B-trees. It will then be shown that synchronizing transactions and their accesses to index records may result in degradation of performance due to the higher traffic volume in the index area. However, if only a portion of the transactions will cause an update in the index area, and that they can be declared before they start, we will in the second subsection show how the HTS algorithm, by assigning the index area to the higher-level data segment in the segment hierarchy, achieves the effect of giving preferential treatment to the index area and allows transactions that update the index area to proceed with lit-

tle interference from transactions that do not have to update the index area. An intuitive example will be given, followed by a quantitative analysis via a simple analytical model that attempts to capture, for comparison purposes, the essence of the contention behavior in this case under two different concurrency control methods: two phase locking and the HTS algorithm.

9.1.1 THE B-TREE INDEX STRUCTURE AND RELATED RESEARCH

The B-tree index structure is a multi-level indexing scheme with dynamic balancing. As shown in Figure 35, in a multi-level indexing scheme, every index record consists of a number of elements of the form <pointer-to-next-level-index-record, high-key>. At the leaf level of the multi-level index tree, the elements in the index record have the form of <pointer-to-data-record,key> and contain pointers to the data records themselves. To search for a data record with a particular key value (or a set of records within a certain range of the key values), a transaction scans the highest level index record, identifies the element in the index record with the lowest high-key value that is higher than the key value being searched for, and advances to the next-level index record by following the pointer contained in that element. It repeats the process until the desired data record is finally located.

Multi-level indices could remain unchanged when data records are inserted, deleted and udpated. This static scheme is suitable for data-

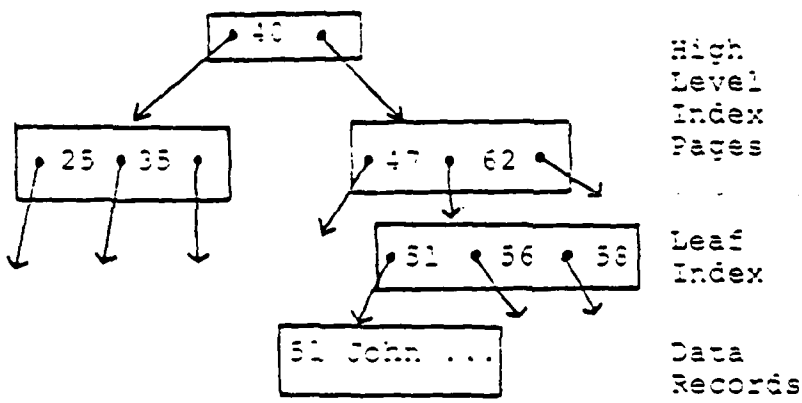


Figure 35. B-tree Indexing Scheme

bases that are not updated very frequently. However, if a database is udpated frequently, then as the database evolves, an initially balanced multi-level index may become grossly unbalanced and cause serious performance degradation in record searching. Therefore under a static scheme the database is subject to periodic reorganization to produce new, balanced indices. An alternative scheme is to introduce dynamic balancing as the database evolves. This gives rise to using the B-tree scheme for organizing indices.

Dynamic indexing is very similar to the static scheme in its data structures. The main difference lies with the additional activity for dynamic balancing associated with each insertion and deletion. In a B-tree indexing scheme, the number of pointer-key elements in an index record is constrained by a parameter k, where

$$k \leq \text{number of pointer-key elements per index record} \leq 2k-1 \qquad (1)$$

Inserting a data record into the database will cause a new pointer-key element to be added to the appropriate leaf index record. However, to enforce (1), adding a new element to an index record where 2k-1 elements are already in the record will cause the record to be split into two. This in turn will cause the higher-level index record to be updated, which may in turn cause the higher-level index record to be split, and the effect may ripple to the highest level index record. Conversely, deletion of a record will cause an element to be deleted from the appro-

priate leaf index record, which may in turn cause a merge of this index record with a neighboring index record to enforce (1), and so on.

Dynamic balancing eliminates the kind of performance degradation a static scheme may suffer from as database evolves, and more importantly, eliminates the need for periodic reorganization of the database. However, the prospect of updating the index records results in the need to synchronize all transactions' accesses to the index area, a requirement not needed in the static scheme since in the static scheme the index area is read-only for all transactions. Since the index area is comparatively small and is to be accessed by a large number of transactions, it is an area of potential threat to level of concurrency in the database. That is, the likelihood that two transactions that do not interfere at the data record level but interfere at the index record level may cause serious degradation in the level of concurrency in the system.

Several locking-based approaches have been proposed to alleviate the problem. Two most recent works of particular relevance are the structural locking approach <Kedem79, Silberchatz80> and the B-link tree approach <Lehman82>. The first approach aims at reducing the amount of time the locks on the high-level index records must be held by each transaction. However, it still involves lock and unlock protocols for each access to the index area, and the scheme becomes fairly complicated

when generalized to a multi-key environment or when a transaction accesses more than one data records (i.e., multi-step transactions.)

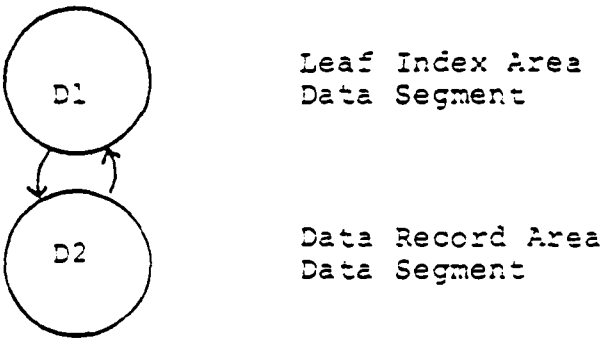
The B-link tree approach is an interesting scheme which recognizes that the basic difficulty of dynamic balancing is the requirement to immediately propagate udpates to the higher-level index records, i.e., an update to the data record and all the associated updates to the index records would have to be atomic. By slightly relaxing this requirement, which basically results in allowing the index structure to be temporarily in an 'imperfect' yet correct state, concurrency can be enhanced.

The scheme that we will describe which employs the HTS algorithm is complementary to the B-link tree scheme. The B-link tree scheme, while being very specialized (i.e., applicable to B-tree operations only) and is not generalized for handling multi-key indexes and multi-step transactions, does reveal a philosophy of optimization of concurrency control methods shared by the HDD approach: delay as much as possible propagation of updates until a more convenient moment while not compromising the goal of presenting the database to all transactions in a correct state. Therefore our scheme can be considered a scheme that is built on top of the general philosophy of the B-link tree scheme and further optimizes the synchronization procedures through a different channel.

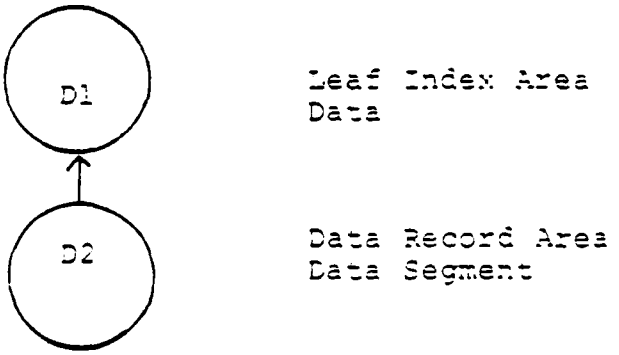
9.1.2 APPLYING THE HTS ALGORITHM

We assume that the B-link tree approach is taken to eliminate the need for an update transaction to be immediately concerned about updating the index records higher than the leaf index records. This can be done by giving the leaf index records temporary fixes after a merge or a split such that, with the old higher level index record, the desired leaf index record could still be obtained. The exact nature of this scheme is fairly involved and is not directly related to the use of the HTS algorithm, and therefore will not be detailed here. However, we assume that the leaf index records and the data records are targets of atomic updates. That is, the effect of updating a data record that results in a leaf index record being updated would be made known to other transactions together with changes in the leaf index record. No partially updated state (i.e., data record updated but the leaf index record not updated, or vice versa) is to be seen by any other transaction. Under this arrangement, we are concerned therefore mainly with the data segment representing the leaf index area and that representing the data record area.

The relevant data segment graph (DSG) and the data segment hierarchy (DSH) in applying the HTS algorithm to the dynamic indexing problem are shown in Figure 36. As shown in the figure, the database is decomposed into two data segments. The higher level data segment (D_1) is the index area, and the lower level data segment (D_2) is the data area. With this database partition, two classes of transactions are identified:



(a) The Data Segment Graph



(b) The Data Segment Hierarchy

Figure 36. The DSG and DSH of the B-tree Index Case

(1) Transaction class 1 (T_1): Transactions that involve updates to one of the indexed fields. These transactions have to write into both the index area and the data area, therefore are transactions that are rooted in D_1 (the higher level data segment) and is the class responsible for the D_1 -> D_2 arc in the data segment graph.

(2) Transaction class 2 (T_2): Transactions that do not involve udpates to any of the indexed fields. These transactions have to write into only the data record area, and need only to read from the index area. Therefore these are transactions that are rooted in D_2 (the only data segment it wirtes into) and is responsible for the arc D_2 -> D_1 in the data segment graph.

Obeying the HTS protocols, accesses to data granules from a transaction will be synchronized by the rules spelled out as follows:

(1) Read(d,t) where d is in D_1 and t is in T_1: Grant access to the most recent version of d before I(t). Timestamp the accessed version of d with a read timestamp with value I(t).

(2) Write(d,t) where d is in D_1 and t is in T_1: Check if the most recent version of d has been stamped by any time greater than I(t). If so, abort t. Otherwise, create a new verion of d with version timestamp I(t).

(3) Read(d,t) where d is in D_2 and t is in T_1: Grant access to the most recent version of d before I(t) + q, where q is a constant predetermined by the system for all transactions in class T_1

such that the probability that a transaction in class T_1 would last longer than q is very small. Timestamp the accessed data version with read timestamp of value I(t) + q.

(4) Write(d,t) where d is in D_2 and t is in T_1: Check if the most recent version of d has been stamped by any time greater than I(t) + q. If so, abort t. Otherwise, create a new verion of d with a version timestamp of I(t) + q.

(5) Read(d,t) where d is in D_1 and t is in T_2: Grant access to the most recent version of d before $A_2^1(I(t))$, i.e., the initiation time of the oldest active transaction in transaction class T_1 at time I(t).

(6) Read(d,t) where d is in D_2 and t is in T_2: Grant access to the most recent version of d before I(t). Timestamp the accessed data version with read timestamp of value I(t).

(7) Write(d,t) where d is in D_2 and t is in T_2: Check if the most recent version of d has been stamped by any time greater than I(t). If so, abort t. Otherwise, create a new verion of d with a version timestamp of I(t).

The above seven categories of accesses are summarized in the following table where each access is given the protocol name used (i.e., Protocol E, H or L.)

9.1.2.1 AN EXAMPLE

Transaction Class & Access Type / Data Segment	T1		T2	
	Read	Write	Read	Write
D1	E	E	H	
D2	L	L	E	E

Figure 37. Summary of protocols used to control accesses in the B-tree case.

As an example, consider the following database and a set of transactions to be executed concurrently:

Data record: <Social Security Number (SSN), income level>.

Keyed field: SSN

t1: Insert record <7048, 10k>.

t2: Update income level of the record with SSN = 6937.

t3: Update income level of the record with SSN = 7059.

A relevant snapshot of the database before these three transactions are run is shown in Figure 38. Let the most recent versions in the snapshot of all records involved in the three transactions be before I(t1), and let I(t1) < I(t2) < I(t3). Now we show a sequence of interleaved execution of these transactions under control of the HTS algorithm:

t1: access record A and leave a read timestamp I(t) with A.

t2: access recrod A.

t3: access record A.

t1: after creating a new record where SSN = 7048 with a version timestamp = I(t1) + q, create a new version of the record A with a version timestamp = I(t1).

t2: create a new version of the record B with version timestamp = I(t2).

t3: create a new version of the record C with version timestamp = I(t3).

All these transactions in our example conflict on the index record A but do not conflict at the data level. However, with the HTS algorithm,

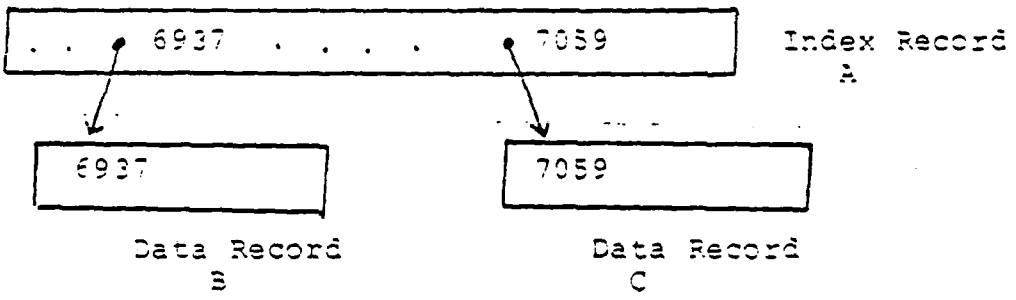


Figure 38. A Snapshot of the relevant initial database state in an example.

all three transactions are processed concurrently without inducing any delay or restart. Moreover, t2 and t3's accesses to the record A do not entail any effort to timestamp the accessed record. The same sequence of actions, if controlled by the conventional MVTS, will result in t2 and t3 leaving read timestamp with record A and also in t1 being aborted and restarted; if controlled by the two phase locking algorithm, will result in t2 and t3 being blocked until t1 is finished in addition to having every record acceses bracketed in the overhead of locking and unlocking.

An intuitive explanation for the above scheme to work is that the HTS algorithm has the net effect of giving a preferential treatment to the index area. As shown in our example, instead of allowing t2 and t3 to directly conflict with t1, which belongs to the class T_1, it made t2 and t3 to appear to t1 as earlier transactions, even though t2 and t3 actually started after t1. Transactions t2 and t3 therefore are simply granted 'earlier' versions and will not leave traces that would have caused t1 to abort. We will in the next subsection present a more analytical explanation of the forces underlying our simple example.

9.1.3 AN ANALYSIS USING A SIMPLE ANALYTICAL MODEL

we now present an analysis to quantify the effect of the HTS algorithm on performance improvement, in terms of level of concurrency, in our dynamic indexing case. The analysis is based on a simple analytical

model of the behavior of transactions synchronized by the two phase locking algorithm and the timestamping algorithm.

9.1.3.1 MODELLING APPROACH

Due to the complexity of the activities involved in a transaction processing system under different concurrency control methods, the present analysis employs a number of assumptions to simplify the problem. The alternative method of study would have been simulation, which would have been both costly and relatively inefficient in providing intuitive pictures. Therefore, for the purpose of obtaining mean values of performance measures, the present analytical approximation approach is chosen. The emphasis of comparison is on concurrency control related overhead, the relavant measures being frequencies of blocking and restarts due to data contention.

The classic computer system performance evaluation studies normally concentrate on the impact of contention for computer system resources, such as CPU and I/O devices, on the performance of the system in terms of response time and throughput. Relatively little attention has been paid to evaluating the impact of *data contention* on the performance of a database system. To some extent, while the notions of a system being CPU-bound or I/O-bound are well established in the field of computer system performance evaluation, the notion of *data-bound*, or *concurrency-bound*, is yet to be established. It is precisely this

dimension of performance limitation that the concurrency control algorithms are to address.

To conentrate on evaluating the impact of data contention rather than contention for other system resources, we propose to conceal the details of the computer systems (CPU, number of I/O devices, etc.) by parameterizing the mean computer service time for each granule request to be a constant 1/u. This implies that the computer system resources are adequately supplied such that marginal increases in load (i.e., size of transactions or number of terminals) will not cause significant contention for computer resources to occur, and therefore maintaining the computer service time per granule at 1/u. Equivalently, we postulate a transaction processing system in which the entire computer system is abstracted into a single 'infinite' server with a constant mean service time per granule access request. We then derive expressions for performance measures for this transaction processing system as functions of parameters of concern to concurrency control, such as transaction size, level of multi-programming, size of read sets and write sets, database size, etc., under each of the concurrency control algorithms to be studied. In short, this approach helps isolating the effect of data contention from that of computer system resource contention and is especially useful in comparative evaluation of different concurrency control methods.

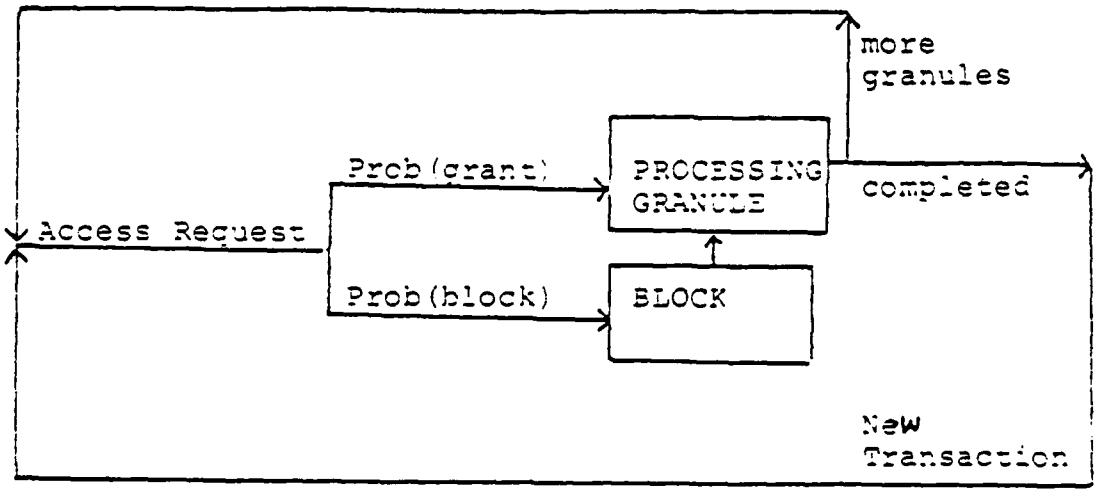
As a short digression, this philosophy of separating data contention from computer system resource contention is not entirely new. Some early work in performance evaluation of concurrency control using simulation (<Spitzer76>, <Muntz77>, <Ries77>, <Ries79>) did not make this distinction. However, the set of simulation studies conducted in (<Lin82a>, <Lin82b>, <Lin82c>) specifically separates the effect of contention for computer systems from that of data contention. In particular, in <Lin82b>, a simulation of the two phase locking concurrency control method is conducted which provides interesting evidences that the performance of the two phase locking method depends very little on the variance of the granule service time. This gives some support for the validity of studying data contention as an entity separate from contention for computer systems. Among the analytical work, <Potier80> studied the effects of two phase locking method on performance, employing a two-level model in which the computer system resource contention is modeled at one level while the data contention and concurrency control are modeled at the other level, and the global performance is derived by combining the two. A similar approach was taken in <Menasce82> for analytically evaluating the performance of the timestamping algorithm. Analytical models that concentrate on data contention are reported in <Shum81>, <Chesnais83>, <Gray81> and <Goodman83>. Our purpose of modelling here, however, is aimed at providing simple formula that are intuitively useful in assessing the impact of the concurrency control methods on performance of a segmented database. Therefore we rely on deriving functional relationships among

system parameters and performance measures at the steady state to shed light on the key differences of the concurrency control methods under study.

Under this approach, we postulate a model of transaction behavior under two phase locking as shown in Figure 39(a) and a model of that under multiversion timestamping in Figure 39(b). In general, a transaction begins by requesting the first granule. Under two phase locking, the request may be blocked and the transaction put in the block queue until it is reactivated. Under multiversion timestamping, however, the request is always granted. Once the request is granted, the transaction spends on the average 1/u units of time in accessing and processing that granule. This time includes all CPU and I/O time needed for processing the granule and waiting time for CPU and I/O devices. The transaction then proceeds to request the second granule, and repeats this process until it is finished. When a transaction has finished processing all its granules, it enters its commit phase whose length is determined by the number of granules the transaction has to write. Under 2PL, the transaction then leaves the system as successfully completed. Under MVTS or HTS, however, a transaction would have to be atomically validated for successful completion. This means that a transaction could be aborted and leaves the system as unsuccessfully finished.

We assume that the transactions in the same class behave in a homogeneous way, meaning that all transactions in the a class T_i access the

Moel for two phase locking:



Model for timestamping:

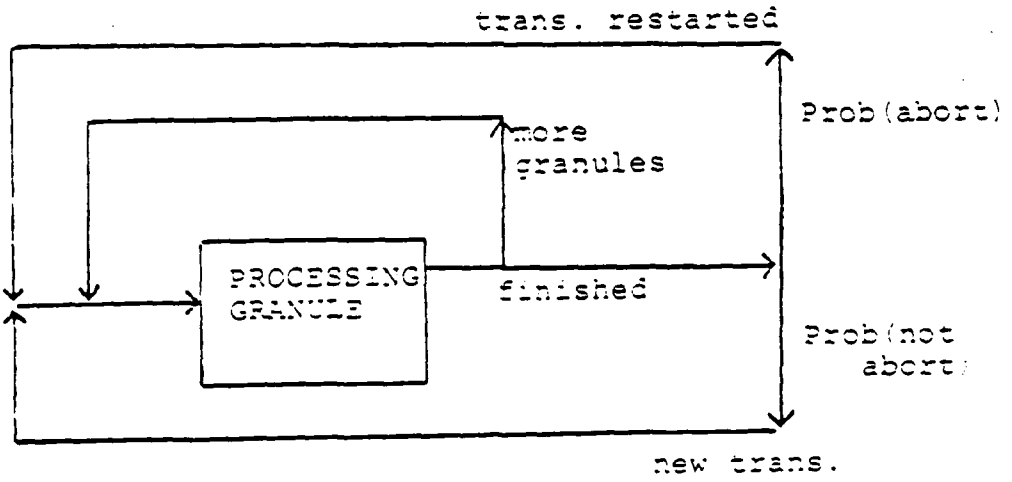


Figure 39. Models of two-phase locking and multi-version timestamping.

same number of data granules in each data segment. We denote the number of granules of data segment D_j accessed (or written) by a transaction in class T_i as $a_{i,j}$ (or $w_{i,j}$) The size of a transaction is equivalent to the total number of granules it accesses. We denote the size of transactions in class T_i as a_i. We also assume that all granules within the same data segment have the same likelihood of being accessed in an access request directed to that data segment.

In modeling the bahavior of the 2PL algorithm, we ignore the deadlock issue to render the model tractable, assuming that its impact on the overall performance is negligible, and we are mainly concentrating on the algorithm's blocking bahavior. This assumption is very likely to be violated in real systems. However, this assumption is in favor of the 2PL approach, and is a conservative assumption when the purpose of the study is to identify scenarios under which HTS can perform better than 2PL.

Finally, we assume that at steady state, the expected number of granules already processed by a transaction in the system is a fixed proportion s of the size of the transaction. (The size of a transaction is given by the parameter a_i for transactions in class T_i.) That is, if one were to pick at random a transaction in class T_i currently in a system at its steady state, then the expected number of granules already accessed by this transaction can be expressed as $s * a_i$, where s is a constant between 0 and 1. This assumption plays an important role in

simplifying the computation of transition probabilities in both the 2PL system and the HTS system.

Given the above, Figure 40 provides a summary listing of the parameters that we will be working with.

9.1.3.2 DERIVATION OF PERFORMANCE MEASURES

TWO PHASE LOCKING MODEL: We will first compute the rate of conflicts in the transaction system under the 2PL algorithm. To do this, one must derive the expected number of granules locked and that write-locked at any time in both D_1 and D_2. Let these numbers be denoted as g_1, g_2, gw_1, gw_2, where g_1 denotes the expected number of granules locked in D_1 and gw_1 denotes the expected number of granules write-locked in D_1, and so on. These numbers can be approximated from the number of transactions in the system and the parameter s, the expected proportion of the number of granules processed by a transaction in the system:

$$g_i = (MP_1 * a_{1,i} + MP_2 * a_{2,i}) * s$$

$$gw_i = (MP_1 * w_{1,i} + MP_2 * w_{2,i}) * s$$

Since we have assumed that data granules within a data segment have the same likelihood of being accessed in any request, the probability that a read request from a transaction in class T_i for a granule in D_i is rejected can be given as:

$$pread(1,i) = (gw_i - w_{1,i} * s) / |D_i|$$

$$= (s / |D_i|) * (MP_1 - 1) * w_{1,i} + MP_2 * w_{2,i})$$

D_1 , D_2 : size (i.e., number of data granules) in data segments D_1 and D_2

$a_{i,j}(w_{i,j})$: number of data granules accessed (written) per transaction in T_i in data segment D_j i,j = 1,2

a_i : total number of granules processed per transaction in class T_i (i.e, transaction size of transactions in calss T_i). i = 1,2.

u : average granule service time (including time spent in waiting for CPU or I/O devices)

s : the estimated average proportion of the total data granules to be processed by a transaction that is already processed by a transaction in the system

MP_1, MP_2 : multi-programming level for transaction classes T_1 and T_2.

(In addition, we use R_1 and R_2 to denote mean response times of transactions in class T_1 and T_2 under 2PL)

Figure 40. Summary of Parameters of the Models

Similarly, the probability that a write request from a transaction in T_i for a granule in D_i is rejected is given as:

$$pwrite(i,i) = (g_i - a_{i,i} * s) / |D_i|$$

Now, to compute the rate of blocking, one must also find an expression for the rate of read and the rate of write requests from each transaction class to each of the data segment. These rates are functions of the throughput of the system. Let the expected response times of transactions in classes T_1 and T_2 be denoted as R_1 and R_2. The throughput of the transactions in class T_i, denoted as $throughput_i$, according to Little's law, is given as

$$throughput_i = MP_i / R_i$$

The rate of read requests from transaction class T_j to data segment D_i is therefore given as

$$rrate(j,i) = throughput_j * a_{j,i} / a_j$$

Similarly one can derive the rate of write request wrate(j,i). Multiply the rate of requests by the probability of conflict per request one derives the rate of conflict, or the rate of blocking, in the system as follows:

Rate of blocking due to accesses to D_i =

$$(s / |D_i|) * (\Sigma(j) \ (pread(j,i) * rrate(j,i) + pwrite(j,i) * rwrite(j,i)))$$

When $|D_1| << |D_2|$, which amounts to saying that the potential contention in D_1 would be much higher than that in D_2, we want to concentrate on the rate of blocking due to accesses to D_1, which is expected to dominate that due to accesses to D_2. Since no transactions in T_2 writes in

D_1, we have $w_{2,1} = 0$. In addition, in our case, $a_1 = a_2$, $a_{1,1} = a_{2,1}$. Therefore, We give the rate of blocking in D_1 in our case as follows:

$$C * (1/R_1) * \{2MP_1*(MP_1 - 1) + (1 + e)*MP_1*MP_2\} \qquad (2PL-1)$$

where $e > 1$ is the ratio between R_1 and R_2, i.e., $e = R_1 / R_2$,

and

$$C = (s * a_{1,1} * w_{1,1}) / (|D_1| * a_1)$$

What is interesting about this expression for approximating the rate of blocking due to contention is D_1 is that it is sensitive to the number of transactions in both classes in the system. Therefore when D_1 is heavily contended for and MP_2 is much greater than MP_1, we expect the system to suffer from a high rate of blocking, much of it would be spent in blocking transactions in class T_2, those transactions that need only to read D_1 and do not even update D_1.

Now we turn to deriving the rate of restarts under the Hierarchical Timestamp Algorithm.

HIERARCHICAL TIMESTAMP MODEL: The major performance characteristics of the HTS algorithm is the rate of abort, i.e., the rate of restarting transactions due to invalidated accesses. We again assume that in the case under study D_1 is much more contended than D_2, and therefore the rate of aborting transactions due to invalidated accesses to granules in D_1 dominates that in D_2. We are, as a result, interested in deriving the rate of aborts due to contention in D_1.

First we derive, for a particular transaction t in class T_1, the expected number of data granules in D_1 that are timestamped by transactions that are started later than t when t enters its atomic validation phase. This number, denoted as $d_1(i)$, divided by the total number of data granules in D_1, is the probability that a write request from t to a data granule in D_1 would be invalidated during the validation phase. Therefore, the probability that a transaction in class T_1 would be invalidated due to requesting to write in D_1 and as a result would be aborted, is given as

$$P_1(i) = 1 - (1 - (d_1(i) / |D_1|))^{w_{1,1}}, \qquad (1)$$

Since in our case, $D_1 \uparrow\!> D_2$, therefore $w_{2,1} = 0$. Therefore, under HTS, only transactions in class T_1 could be aborted due to invalidated write requests for granules in D_1, while transactions in T_2 will never be aborted due to contention in D_1.

Now we proceed to derive $d_1(1)$, the expected number of data granules timestamped by times later than I(t) when transaction t in T_1 enters its atomic validation phase. To compute this, we first derive the expected elapse time between the initiation time of t in T_1 and the time when t enters its atomic validation phase. This time, however, is simply the total granule processing time of the transaction, a_1/u, since under the HTS algorithm no time of a transaction is spent in the block state.

Next we derive the expected number of transactions in class T_1 that are started during this period of time. This is given as $a_1/u * (MP_1$

-1) / $((a_i + w_i)/u)$, since MP_i / $((a_i + w_i)/u)$ gives the rate of starting transactions in T_i. Finally, we derive $d_i(1)$, the expected number of granules in D_i timestamped by later transactions when t in T_i enters its validation phase. Since, under HTS, no transactions in class T_2 would timestamp data elements in D_i, (recall that all transactions in T_2 use protocol H to access D_i, which does not require any timestamping), $d_i(1)$ will depend solely on the rate of starting transactions in T_i and the expected stage at which these transactions started after t would be in when t enters its validation phase. The latter is determined by the parameter s. We therefore derive an approximation of $d_i(1)$ as follows:

$$d_i(1) \simeq (a_i/u) * (MP_i/((a_i+w_i)/u) * s * a_{i,1} \qquad (2)$$

This formula in fact gives an upperbound for $d_i(1)$ since it makes the assumption that there is no overlap of granules timetamped by the younger transactions. Therefore we must keep in mind that the resulting measures of rate of abort would tend to be overstated.

Plugging (2) into (1), we derive the probability that a transaction in T_i would be aborted due to invalidated write request for granules in D_i:

$$P_i(1) = 1 - \{1 - ((MP_i - 1)*s*a_{i,1}*a_i)/(|D_i| * (a_i+w_i))\}^{w_i \cdot 1}$$

The rate of aborting transactions due to contended accesses to D_i is therefore $P_i(1)$ multiplied by the rate of starting transactions in class T_i, and can be approximated by the following upperbound:

$$C * (a_i/(a_i+w_i))^2 * u * (MP_i*(MP_i - 1)) \qquad (HTS\text{-}1)$$

where C is defined the same way as in (2PL-1).

Compare (HTS-1) with (2PL-1), one discovers the difference that (HTS-1) does not depend on the level of multi-programming on throughput of the transactions in class T_2. Therefore, if the rate of blocking due to contention in D_1 is the source of performance degradation under 2PL and the rate of restarting is that under HTS, HTS clearly has the potential of offering an advantage over 2PL, especially in cases where the throughput requirement for transactions in class T_2 is much higher than that in T_1.

CONCLUSION: In sum, we have adopted a very simple analytical model to capture the essence of the performance of the 2-data segment case under the two phase locking algorithm and the hierarchical timestamp algorithm. The model makes assumptions that are in general in favor of the 2PL approach and discriminates against the HTS approach. The purpose is to demonstrate the effect of the HTS algorithm in relieving contention in the higher data segment, presumably the much more heavily contended data segment in our case. The result shows that, in general, the rate of blocking under 2PL is proportional to $MP_1^2 + MP_1 * MP_2$. In contrast, the rate of abort under HTS is proportional only to MP_1^2. While the absolute values depend on other parameters, one can readily conclude that HTS is preferred when $MP_2 >> MP_1$.

9.2 DESIGN OF A BANKING DATABASE

In this section we apply the hierarchical database decomposition approach to the design of a banking application. As discussed before, the hierarchical approach to structuring the database components offer advantages additional to reducing data contention and synchronization overhead. It also encourages transaction and database designs that take advantage of the pipelining operations that may be intrinsic to an application, and results in designs that are more efficient, better structured and more resistant to localized failures. While the main purpose of the current case study is to demonstrate the existence of applications that can be structured hierarchically and therefore are candidate applications where the hierarchical timestamping algorithm can be employed to reduce contention and synchronization overhead, this case is also chosen so as to exemplify the structuring effect of the HDD approach.

A synopsis of this section is as follows. We will in the first part provide a description of the current operation of the banking system under study. The current system, considered one of the most advanced in what it does and is quite sophisticated and complex, is under review by the bank to find ways to improve on it so as to better meet the demand on the system as one of the leading electronic banking systems. The current system is composed of a number of independent systems that have been developed separately for different bank product offerings, such as loans, funds transfer, cash management, etc. Due to the fact that the current system is not intrinsically an integrated system, it is consid-

ered disadvantaged for dealing with integrated customer services and new product offerings and with growth in demand in services.

To provide a basis for integration, a new architectural design of the system has been proposed. This new design fully exploits the database system technology in order to separate processing from data, so that the database could serve as an integrated source of information for all types of existing and future processing. This conceptual model of the new system will be briefly described in the secnd part of this section.

Taking the conceptual model as the future system architecture, this banking system, in its integrated form, i.e., as a database applicaiton, appears to exhibit a hierarchical structure where the HDD approach to database concurrency control can be applied. This hirearchical structure can be derived from the types of operations and transactions supported by the current system. In other words, if we were to pull all the data now residing, and perhaps duplicated, in the various systems and perform an analysis on the access and update behavior of the transactions on these data, one can readily identify a hierarchical strucutre to the database where segments of a higher level are read, but need not be updated, by transactions that update segments of a lower level. This hierarchical structure of the database and its corresponding transaction types and classes constitute a proposed high-level database design for the database of the new banking system, and will be described in the third part of this section. This design has the salient feature of

achieving the effect of data integration without concerns over increased data contention and synchronization overhead. The section is concluded with the advantages of the proposed design in its final part.

9.2.1 CURRENT SYSTEM

This description of the current system is mostly extratcted from <Document-A> and from site visits and verbal follow-ups. The purpose of this description is to provide an overview of the the structure of the current configuration and the types of processing in the current system and use it as a basis for designig an integrated system in which information resources are concentrated in a well-structured database.

The current system is composed of various systems that support communication, financial processing (FP) and information processing (IP). Communication is performed by a number of communication networks internal or external to the bank entity under study. The purpose of these communication networks is to route messages, requests and activity information among the financial processing systems, the information processing systems and the external world. Since our focus of study is the information resources and processing, we will not be concerned with the detailed nature of the communication networks.

Financial processing refers to the processing of customers' requests for services, namely, transactions. It is itself supported by several

different systems, each of which supports a particular type of financial service provided by the bank. Currently, there are six systems in this group:

(1) The Funds Transfer System

(2) The Loan System

(3) The Trade Service System (mostly for processing letters of credit)

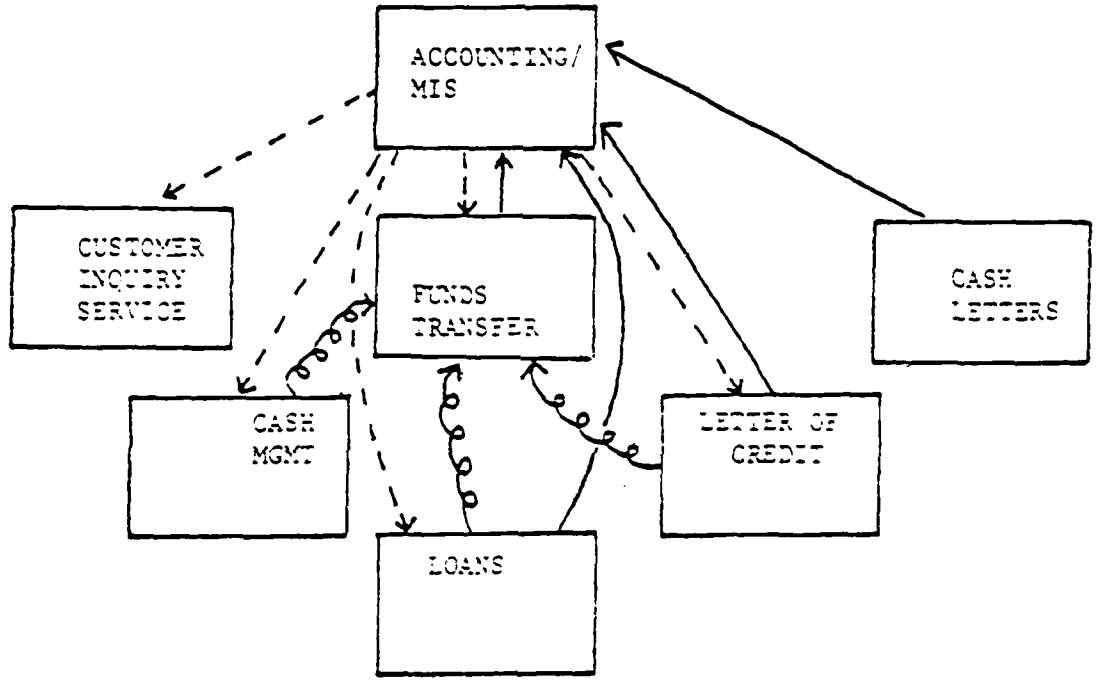
(4) The Cash Letter System

(5) The Cash Management System

(6) The Customer Inquiry and Investigation System

Among these systems, the funds transfer system provides a basic service that most other systems would use, i.e., transaction processing in other systems may result in funds transfer requests being sent to the Funds Transfer System.

All the FP systems also produce posting requests to be handed over to information processing (IP) at the end of an accounting cycle which will use these posting information to compile account status. At the begining of an accounting cycle, the FP systems also receive updated account data from IP to be used as the basis for transaction processing during that accounting cycle. The logical relationship among these financial processing systems and IP is shown in Figure 41.



-----→ : daily update

——→ : posting/transaction log

~~~→ : funds transfer requests

Figure 41. The Logical Relationship Among FP Systems and IP
~~~

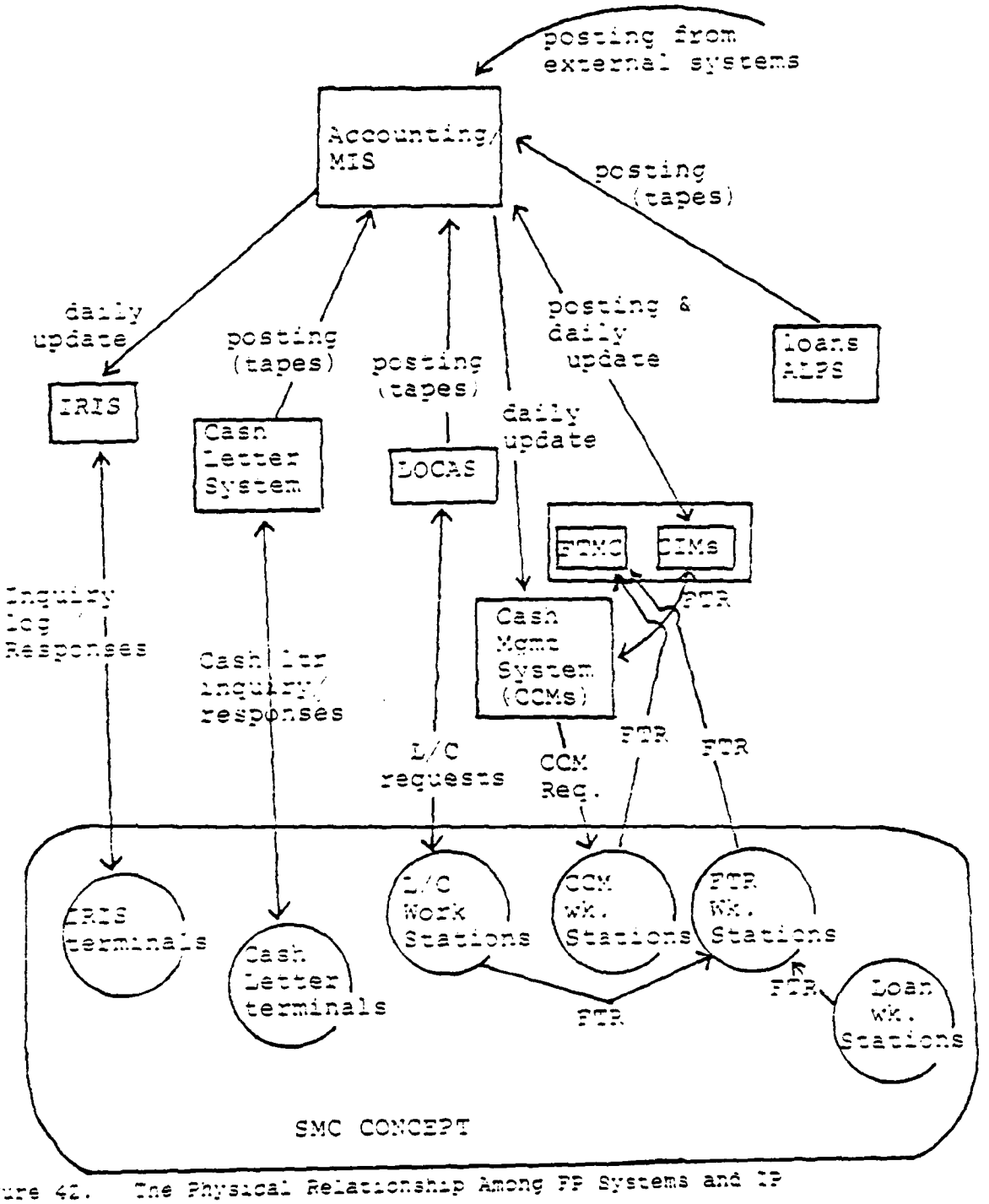


Figure 42. The Physical Relationship Among FP Systems and IP

The various financial processing systems are currently implemented as separate entities. That is, every system is a 'sealed' system, communicating with the world external to the system mainly via its own terminals. Data files used by a system are tightly coupled with the processing component of that system. There exists no physical shared data files across systems. The current physical relationship among these financial processing systems and IP is shown in Figure 42.

The physical configuration of the current system uses the Service Managment Center (SMC) approach to remedy the problems arising from a lack of direct communication among the systems. In an SMC, terminals that are connected to various systems are pulled together in one site to form an uniform interface to the customer requests. Customer inquiries and requests come to this site through phone lines or other communication networks and are interpreted and processed by an operator who would in turn route the request to the appropriate terminal. Use of the service management centers is a first step towards the realization of an integrated customer service.

Information processing (IP) refers to processing of custmer and bank account information for accounting and MIS purposes. It consists of journal, customer ledger and bank ledger processing, proofing - a process that checks for consistency among journal entries and ledgers and if necessary generates repair postings for corrections, MIS processing, such as account profitability data compilation, and risk control infor-

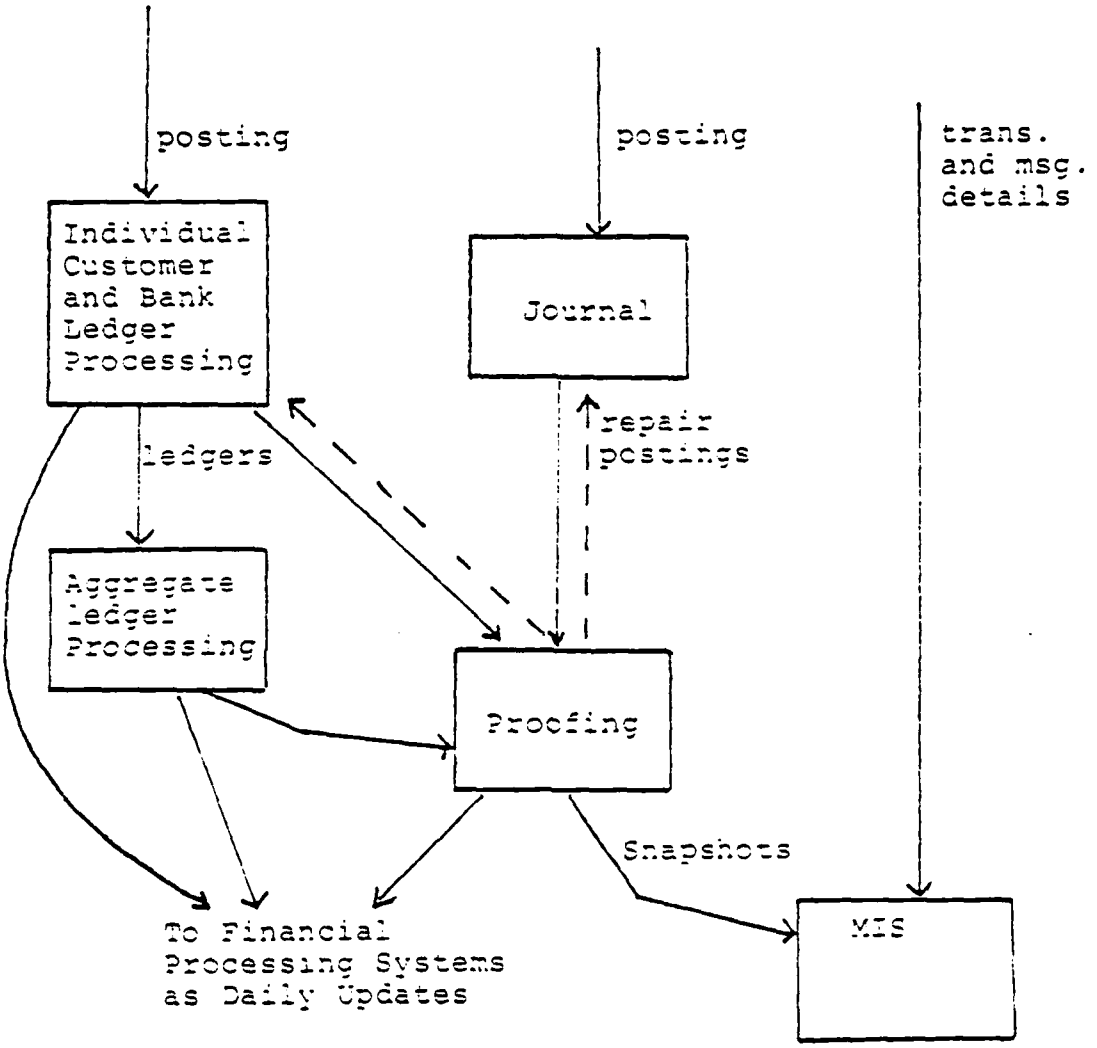


Figure 43. Activities Within Information Processing (IP)

mation processing, which computes the bank's risk exposure to various level of aggregated customer entites. IP accepts posting and transaction information from the financial processing systems. The customer account status information obtained in IP is then fed back to the financial processing systems. The IP activities are diagrammically shown in Figure 43.

Currently, IP is basically a batch processing system which accepts postings from and feeds account status information to various other systems via tape handoffs or hardcopies. IP is implemented on computer systems physically separate from financial processing systems, and can also be considered as a 'sealed' system.

9.2.2 MOTIVATION FOR AN INTEGRATED SYSTEM

The configuration of the current system is a result of both technical considerations (e.g., see <Matteis79>, <Reef79>) and organizational considerations. Segmenting systems by product line makes it easier for individual organization units to plan, develop and control the system that concerns the product line for which the organizational unit is responsible. However, changing needs have made it essential that segmented systems be strategically integrated. The SMC facility currently employed by the bank is a way to emulate an integration of systems that are currently separate.

In <Madnick83>, several scenarios for the need to integrate historically segregated systems were discussed. Extending discussions presented in that paper and the situations faced by the bank in our case study, we summarize the motivation behind integration as follows:

(1) Separate systems make it difficult to provide a consistent interface to a customer who purchases various financial products provided by the bank. For example, the status of a customer across all financial products is not readily available, since it has to be separately pulled out from various systems. An inquiry into a customer transaction that spans product lines can not be serviced efficiently.

(2) Dispersed information makes it awkward to introduce new, multi-service products quickly. For example, the cash management service now provided by the bank which enables the customers to query their account status directly from their terminals cannot access the current account balances in the funds transfer system. Therefore the account balances made available to the customers are information compiled from the previous accounting cycle. Further making available informatin now residing in other financial processing and information processing systems to customers at their terminals would also prove to be time-consuming.

(3) In general, various manual intervention now required to periodically obtain information from one system for processing in

another is both costly and may prove to be a limiting factor for potential future growth.

(4) There are potential problems associated with squeezing the current volume of information processing work in a batch processing window, a window whose size depends on how long the financial systems have to be kept on-line. In the current system, there are only occasional misses since the batch window is still relatively large. However, one must begin to plan for the shrinking of this window as user demands it and an increase in system load (e.g., volume of transactions). Efforts that move information processing such as ledger updating on-line so that it can be performed concurrent with financial transactions will drastically reduce the amount of work that must be squeezed in the batch window. This amounts to the need for IP to directly access transaction information and posting requests generated by TP on a real-time basis.

(5) For the purpose of risk control, events that are taking place in another system within the current accounting cycle may also be important. However, this information is not available for decision making when daily accounting processing is the only way for a system to become aware of events captured by another system.

In sum, an integrated system offers important strategic advantages over the currently segregated configuration. As a result, an architectural design of an integrated system has been proposed <Document-C>. In

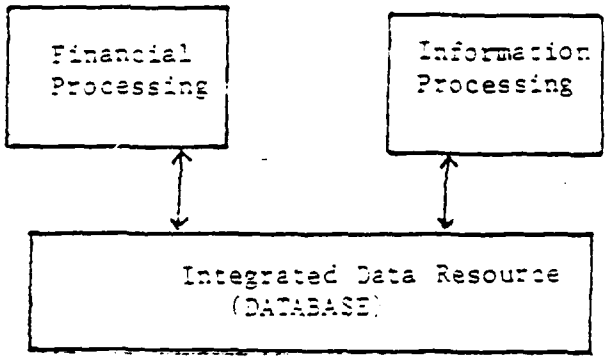


Figure 44. Integrating Separate Systems via Integrated Data Resource

essence, the proposal suggests that the new system be designed such that information resources are separated from processing, i.e., transaction and information processing of all types is to be implemented on top of a shared information resources of the bank entity. Figure Figure 44 captures the spirit of this proposed architecture.

9.2.3 DESIGNING THE STRUCTURE OF THE INTEGRATED INFORMATION RESOURCE

The above paragraphs provide a background against which a study of the structure of the integrated information resources in our case is attempted. The goal of this study is two-fold. First, a structure of the information resource, combined with the HDD approach to database concurrency control, can help integrating data now dispersed in separate systems without increasing data contention. Secondly, this structure may be used as an architecture for guiding future detailed database design activity.

The study is guided by the simple idea that database design is inevitably interlocked with transaction design, i.e., DB design depends on how data is accessed, used and updated by transactions. In serving real-time transaction processing, every effort should be made to make transactions as short as possible so that resources can be freed up quickly and response times optimized. This amounts to deferring as much as possible processing that is not absolutely necessarily real-time. Much of a database often contains derived information of one level or

another. Not requiring real-time transaction processing to be also responsible for updating all levels of derived data not only reduces the size of the transactions, but also has the potential of drastically reducing synchronization overhead if synchronization methods that take advantage of this feature is employed.

As a contrasting example, suppose the database application system is designed in such a way that every event made known to the system would cause all elements in the database that might be affected by the event to be updated. This means that, for example, when a funds transfer transaction is processed in a banking system, in addition to updating the account balances of the two parties involved in the transfer, customer ledger, bank ledger, bank journal, total bank assets, account balances aggregated by various units, etc., must also be updated before the transaction commits. This extremity inevitably results in the transaction running for a long time, tying up critical data resources and causing many other transactions to wait or to be aborted. Obviously in reality an organization can afford and often do resort to a great deal of deferred processing so as to limit the scope of one transaction and produce more efficient processing.

The study begins with an examination of the major transaction and processing types in the current system. Combining the current behavior of the transactions and a document concerning general service requirements and functional primitives <Document-F>, we identify, for each type

of transaction or processing, the data elemnts or files in the integrated information resources that would be accessed or updated. The access patterns of these transactions are then analyzed for identification of a hierarchical segmentation structure that will help in reducing synchronization overhead.

The left two columns in Figure 45 is a list of the transaction and processing types identified for the current system along with their data requirements. Two major observations are made which lead to the construction of a hierarchical segmentation of the database to be presented in a later paragraph:

(1) There are three types of 'risk control information' that are to be shared by various types of transaction processing:

   a. Aggregate account balances by region, country, conglomerate and industry.

   b. Customer risk control parameters such as overdraft limit and credit rating.

   c. Customer demand deposit account balances.

   All three types of information are dynamically changing. However, the aggregate account balances need to be updated, or refreshed, only periodically, and can tolerate relatively long delays - even through the capability of the system to produce the most updated aggregate account balances (e.g., total bank loan exposure to Iran) real-time on demand is extremely desirable. The customer risk control parameters are subject to update

TRANSACTION TYPE	ACCESS PATTERN	ROOTED D.S.*
Customer risk control parameter update	Read: Customer risk control parameter write: Customer risk control parameter	D_1
Funds Transfer (FTR)	R: Customer risk control parameter, Customer DDA balance, Aggregate account risk control parameter W: Customer DDA balance, FTR records	D_3
Loan transaction	R: Customer risk control parameter, Aggregate account risk control parameter, Customer loan balance W: Customer loan balance, loan records Trigger: FTR	D_2
Letter of Credit	R: Customer risk control parameter, Customer L/C balance W: Customer L/C balance, L/C records Trigger: FTR	D_4
Cash Mgmt request	R: Customer DDA balance W: Cash mgmt request records T: FTR	D_4
Cash letter transaction	W: Cash letter posting	D_5
External transaction	W: External system transaction records	D_6
Journal Ledger update	R: FTR records, loan records, L/C records, External system transaction records, Ledger W: Journal/Ledger	D_7
Customer accounting	R: Journal/Ledger W: Customer account statement	D_8
Bank accounting	R: Journal/Ledger W: Bank account statement	D_9
Proofing	R: Journal/Ledger W: Account snapshots	D_{10}
Historical D.B. update - account snapshots	R: Account snapshots W: Historical D.B.	D_{11}
Historical D.B. update - transaction history	R: FTR records, loan records, L/C records, External system transaction records W: Historical D.B.	D_{11}

(Continued from previous page)

TRANSACTION TYPES	ACCESS PATTERN	ROOTED D.S.*
Aggregate accounting	R: Account snapshots W: Aggregate account snapshots	D_{12}
MIS processing	R: Historical D.B. W: MIS statements	D_{13}
Customer inquiry	R: Historical D.B. W: Customer inquiry log	D_{11}

*: Based on proposed data segment hierarchy as described in text and in the data segment hierarchy figure

Figure 48. Types of transactions/processing and their rooting distribution based on a proposed data segment hierarchy.

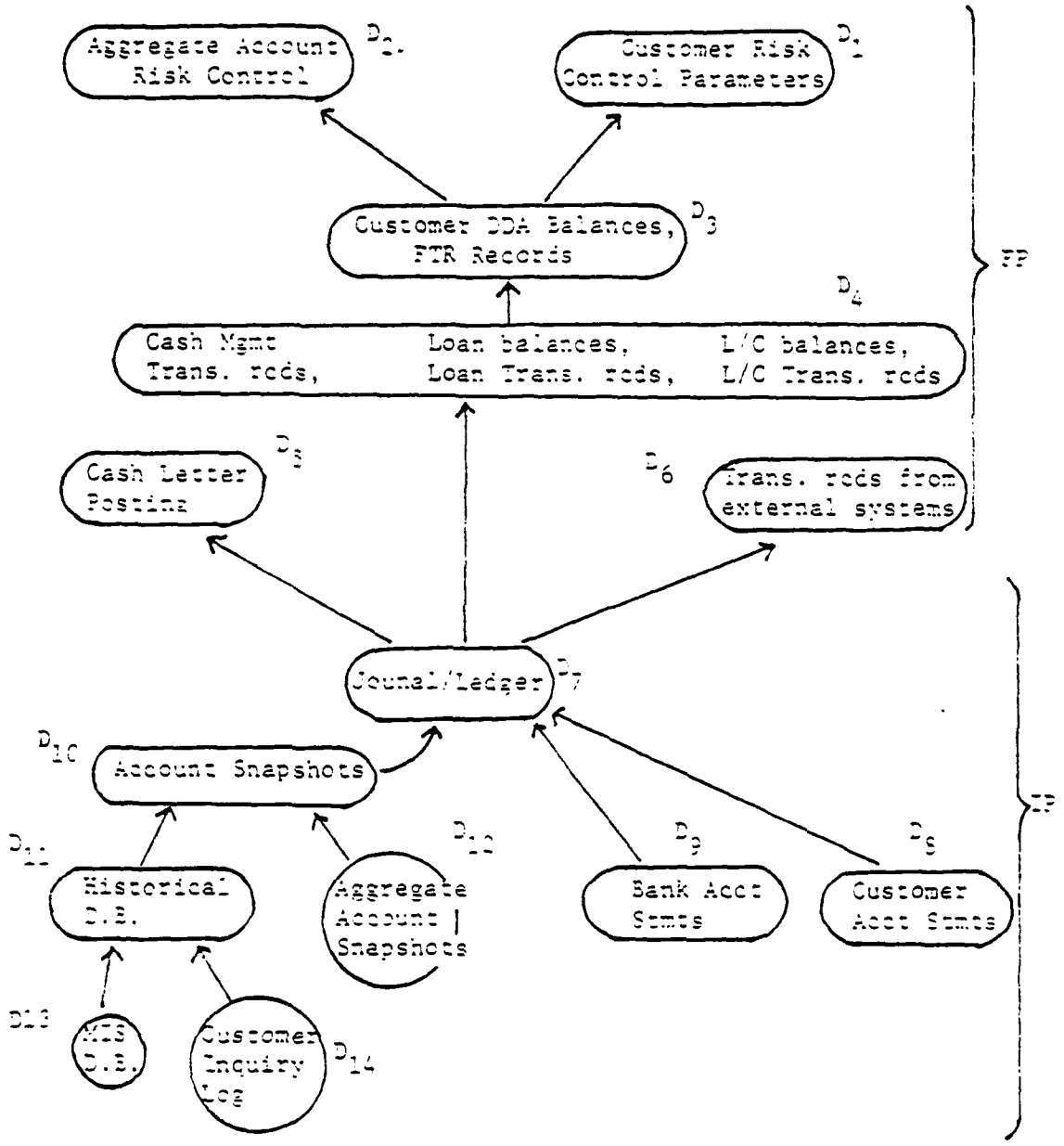


Figure 46. Hierarchical database segmentation of the banking database(the proposed data segment hierarchy).

on-line by bank personnel based on customer requests and other information. The customer demand deposit account balances, on the other hand, are updated very frequently, triggered by the high-volume funds transfer transactions. Since all three types of risk control information are accessed by real-time financial transactions and subject to on-line update, a hierarchical structure of the database of concern to these financial processing which minimizes interference among the different types of transactions demanding these information is desirable. In particular, the customer demand deposit account balances, which are subject to high traffic update from funds transfer processing and high traffic access from most other types of financial transaction processing, is designated at a higher level than the data updated by other FP transaction processing. This design makes DDA current balances available to many other financial transaction processing components on a real-time basis without jeopardizing its concurrent availability to the funds transfer transactions themselves. A non-cyclic database partition of the portion of the database accessed by the FP processing components is shown in Figure 46.

(2) Information processing of the banking system is triggered by transaction processing. For reasons outlined in the last subsection, it is desirable that IP be performed on-line concurrent with processing of FP transactions. However, it is also clear that updating ledgers and journals, etc., does not have to be

executed as an integral part of the financial transactions that have triggered it. In other words, a financial transaction can be committed to the database before the ledger and journal updates triggered by it are committed. This amounts to designing processes in IP as database transactions that are separate from FP but nevertheless need to read the transaction records produced by the FP transactions. At the end of an accounting cycle, several other types of IP transactions are also triggered to produce accounting and MIS statements. These latter processes amount to database transactions that read from the ledger and journal data segment to produce other derived information in the database. A hierarchy of accesses to these information is therefore exhibited by these processing activities, and the nature of it in our case results in a non-cyclic database partition of the portion of the database of concern to the IP processing, which is shown in Figure 46.

In sum, the study results in a hierarchical segmentation of the integrated information resources to be used in the integrated banking system. The data segment graph (DSG) and the data segment hierarchy (DSH) of this segmentation are shown in Figure 47 and Figure 46 on page 248. Transaction classification corresponding to this segmentation is summarized in the last column of Figure 45 on page 247.

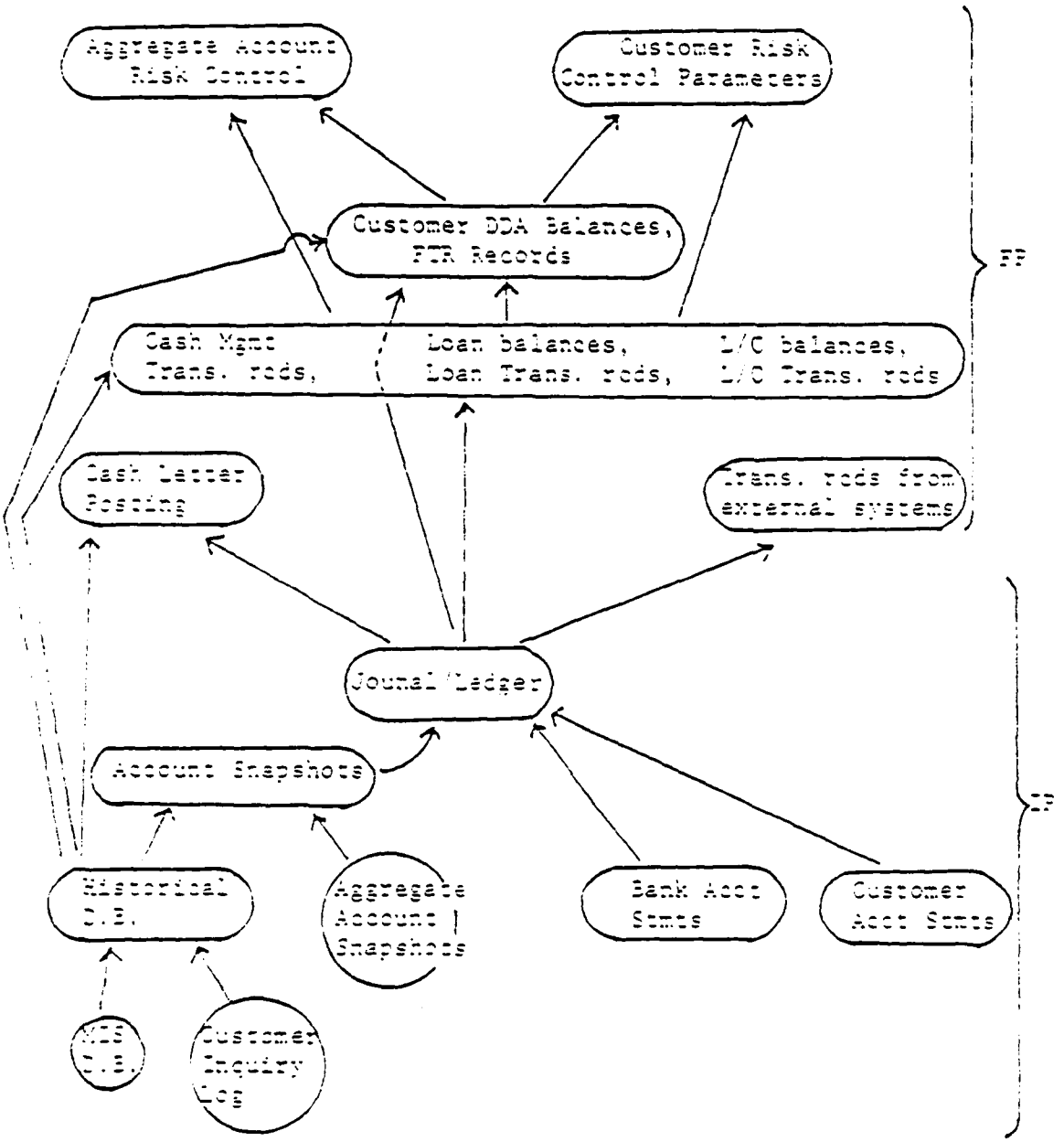


Figure 47. The Data Segment Graph of the propsoed design

9.2.4 ADVANTAGES OF THE PROPOSED HIERARCHICAL SEGMENTATION OF THE DATABASE

We have in this section provided a case study that demonstrates the existence of hierarchical structure of database applications that can benefit from the HDD approach to concurrency control. We list the following advantages of the proposed hierarchical segmentation of the banking database under study:

(1) Automate all manual inter-system communication.

(2) Eliminate duplication of customer information for risk control and the historical database for inquiry.

(3) Enable the system to operate in an environment where on-line processing is close to 24-hour per day and the batch window is correspondingly squeezed.

(4) Enable the system to operate with much more timely information.

(5) Achieve the above through integrating information now residing in separate systems into a shared data rosources with virtually no increase in data contention and very little increase in synchronization overhead.

9.3 FUNCTIONAL DECOMPOSITION IN A MULTI-PROCESSOR DATABASE COMPUTER

In this section we apply the HDD approach to concurrency control to the design of a database computer aimed at large-scale multiprocessing. The first part of this section introduces the architecture of the database computer INFOPLEX under study, together with the motivation for the chosen architecture in comparison with the approaches taken in other database and transaction oriented multiprocessor systems. The second part describes how the HDD approach is closely related to the design of the Functional Hierarchy of INFOPLEX.

9.3.1 INFOPLEX DATABASE COMPUTER ARCHITECTURE

INFOPLEX consists of a Storage Hierarchy providing a very large data storage system, and a Functional Hierarchy, built on top of the Storage Hiearchy, responsible for providing database management functions. Previous work in INFOPLEX has centered around the overall hardware architecture of INFOPLEX <Madnick79> and the design and evaluation of Storage Hierarchy <Lam79>. A preliminary design for the Functional Hierarchy is reported in <Hsu80, To82, Hsu82>. As shown in Figure 48, the Storage Hierarchy is comprised of levels of storage devices with various performance and cost features. The high-performance devices, such as cache memory and main memory, are placed on the top (i.e., the highest level of the hierarchy,) while low-performance devices such as mass storage systems are placed at the bottom. Storage Hierarchy supports Functional Hierarchy by providing a very large virtual address space with a small average access time. The actual structure of Storage Hierarchy and move-

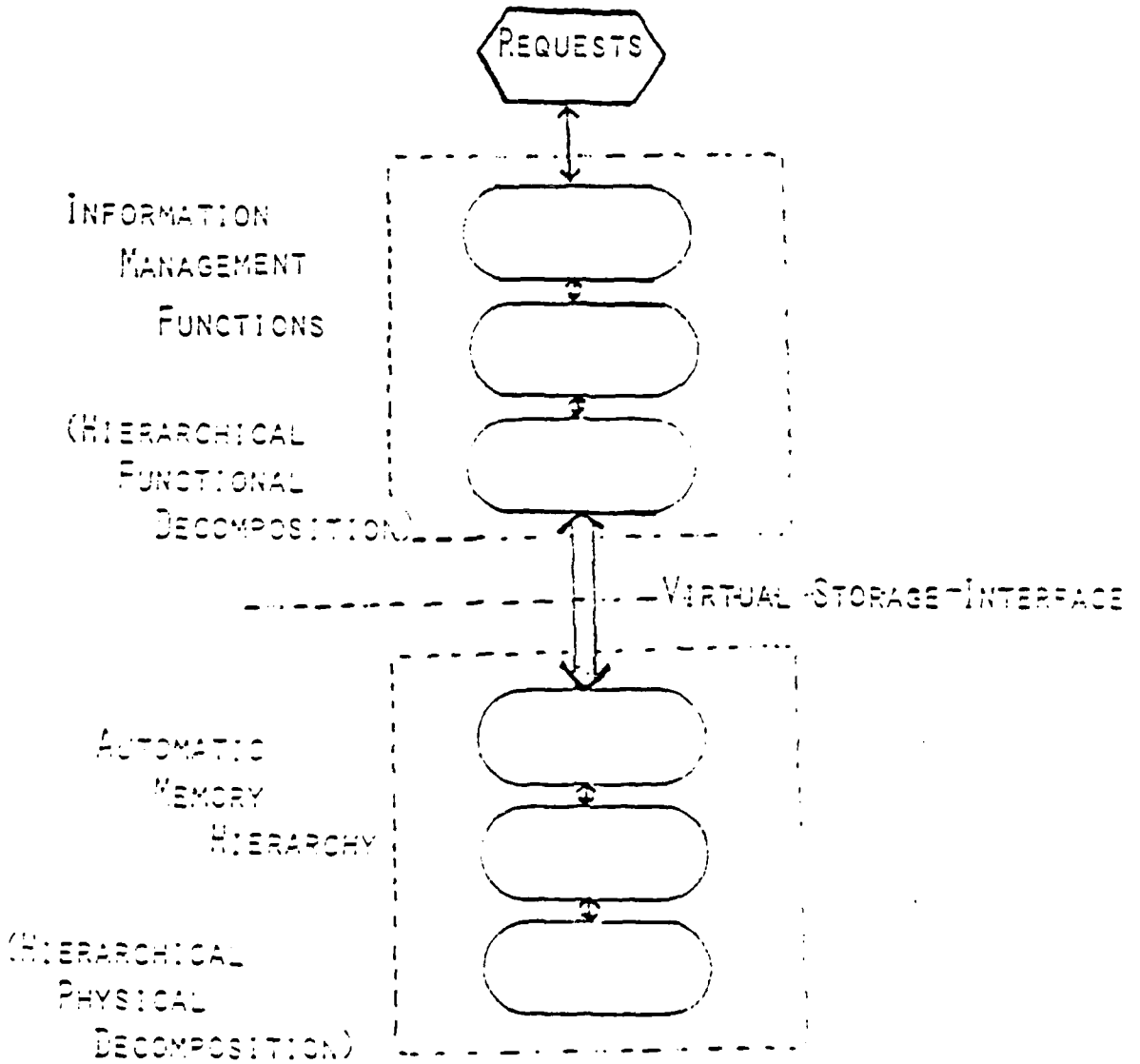


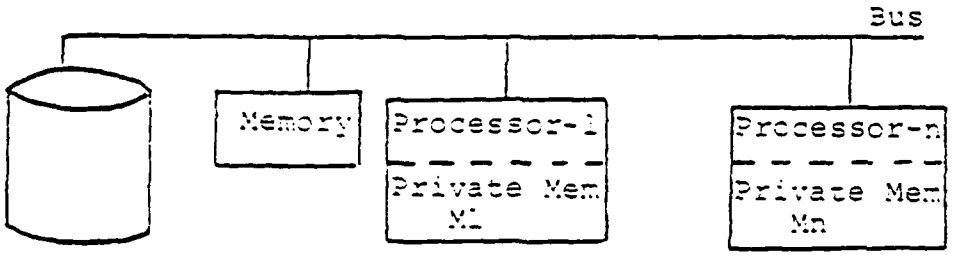
Figure 48. Architecture of INFOPLEX database computer.

ment of data between levels within the storage system are hidden from Functional Hierarchy. Requests for data blocks are made to the highest level of the Storage Hierarchy, and data is moved automatically between levels of the Storage Hierarchy.

The Functional Hierarchy, on the other hand, is a set of microprocessors and memory modules that are structured into a number of processing clusters. Each processing cluster is called a 'level' of Functional Hierarchy, and it essentially implements a 'layer' of the hierarchically decomposed DBMS functionalities. In this section we will concentrate on the design considerations of the architecture of the Functional Hierarchy from the concurrency control perspective.

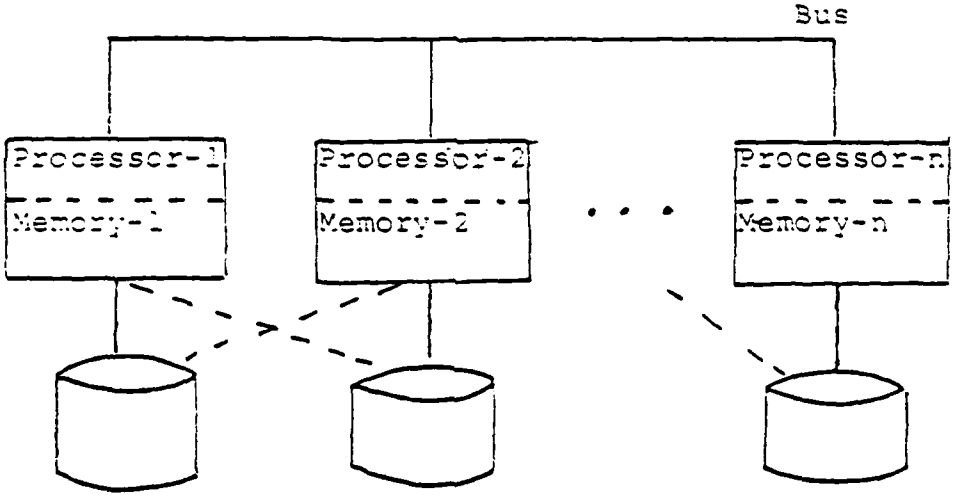
9.3.2 THE STRUCTURED CLUSTERING APPROACH TO MULTIPROCESSING

The conventional approaches to memory organization taken in the transaction- and database-oriented multiprocessing systems can be classified into three categories. The first is the tightly coupling approach, exemplified by such systems as Synapse System. In this approach, processors share the same address space and communicate with one another through shared memory. However, to alleviate contention on the linear communication bus as shown in Figure 49, private memory moudules are attached to processors as cache memory. Processors obtain data from their own cache memory and would need to access the shared memory only when the data needed is not in the cache.



(a)

- M1, ..., Mn and M share the same address space
- Example: Synapse systems



(b)

- Memory-1, Memory-2,..., Memory-n are separate address spaces
- Example: Stratus System

Figure 48. The tightly coupling and the segregated approaches to multiprocessing.

The tightly coupling approach is simple and quite efficient. However, it gives rise to the cache consistency problem which is a synchronization problem more primitive than the database transaction synchronization problem. The Synapse system uses hardware locks to deal with this problem, locking up a page and making it unavailable for being loaded into a processor's cache if the page is currently in the cache of another processor's. By making the hardware architecture completely transparent to any software other than the operating system, the database concurrency control problem is not dealt with using any special technique. The disadvantage of this tightly coupling approach is that it has to employ two layers of synchronization, one dealing with the cache consistency issue and one dealing with the logical transaction consistency issue. This aspect combined with a lack of structure in the architecture makes it potentially unfit for a very high degree of multiprocessing due to contention among processors for the shared memory pages and data elements.

The second approach is the segregated system approach as exemplifed by such systems as Tandem System and Stratus System. In this approach, as shown in Figure 49, every processor is equipped with its own memory and I/O capability. There is no shared memory among processing elements. Centralized dispatch is normally the method used for distributing input jobs among processing elements, and control over shared data is also centralized in the processing element that runs the database management system. This approach does away with the cache consistency

problem. However, centralizing database management activities within one processing element limits the capability of this architecture to meet the demand typically expected on a very large database computer. This approach, like the tightly coupling approach, is oriented towards more general purpose multiprocessing system than a database computer.

The third approach is the functional specialization approach exemplified by such experimental systems as DBC <Hsiao79>. This approach emphasizes specialized hardware for certain types of processing within the database system in order to increase throughput. The drawback of such approach lies with the need for special hardware, the cost of which may prove to be an inhibiting factor. However, the basic philosophy that DBMS performs functions that possess structural properties that can be taken advantage of is a valuable one.

In sum, the first two approaches, while flexible and more suitable for running application software, does not provide for an environment for accommodating a large number (in the order of hundreds) of microprocessors. This is due to the hardware limitation of supporting a large number of processors on a linear bus and, more importantly, the software limitation of data contention. They do not attempt to exploit in the multi-processor architecture structural and functional modularity that may be inherent in database application systems. The third approach, on the other hand, tends to tie database logic with processor

technology, and makes it more costly and less adaptable to changes in either area.

In contrast, INFOPLEX uses the structured clustering approach to multiprocessing. In this technique, a large number of general purpose microprocessors are grouped into clusters called 'levels'. As shown in Figure 50, processors within a level are tightly coupled using the conventional method of a shared bus (local bus) and shared memory modules. Intra-level cache consistency problem is avoided by allowing only read-only pages (e.g., pure code) to be resident in private cache attached to each processor. Multiple levels are in turn linearly coupled via a global bus, but no shared memory exists between levels. In other words, each level has its own address space and does not share its address space with any other level, therefore the inter-level memory consistency problem is avoided. Inter-level communication strictly takes the form of message passing which is performed through hardware (i.e, the gateway controller) without resorting to inter-level shared memory.

Each level is also supported by the Storage Hierarchy. The Storage Hierarchy implements a segmented address space each segment of which is accessible by and only by a particular level. Therefore, the Storage Hierarchy performs the function of a common mass storage utility for all levels without violating the rule that no inter-level shared memory is allowed.

Figure 50. A Level of the Functional Hierarchy of INFOPLEX

With this architecture, the DBMS functionalities are decomposed into layers of tasks, each layer to be assigned to a level. Tasks within a layer are defined through the layer's interfaces to adjacent layers, and chagnes within a layer are relatively well contained.

The structured clustering approach offers several advantages. First, general-purpose microprocessors are adequate as processing elements within each cluster and therefore specialized hardware is not required. Second, it provides a framework for supporting a large number of processors in order to provide for the level of parallelism needed to implement a very large database. Most importantly, this design specifically does away with the memory inconsistency problem. Third, it is a structured approach to implementing large database computer, possessing the potential of resulting in a more modular and more reliable design. We will now turn to the database transaction concurrency control issue in this particular architecture.

9.3.3 CONCURRENCY CONTROL IN AN DBM ARCHITECTURE WITH STRUCTURED CLUSTERING

One aspect of reasearch in the INFOPLEX Functional Hierarchy is a methodology for functional decomposition. The goal of functional decomposition is to distribute functionalities of a database and transaction processing system among the levels of the Functional Hierarchy so as to minimize inter-level interferences. Several considerations must be tak-

en into the specification of a methodology. First, as specified by the architecture, inter-level communication strictly takes the form of message passing and no shared data is allowed between levels. This imposes a constraint on the acceptable decompositions that modules sharing the same data files or tables are better clustered together into one level. Otherwise, these modules would incur a heavy overhead of message passing to get jobs done. Other considerations for decomposition include frequencies and sizes of inter-module communication. For those modules that communicate frequently and involve large quantity of data being passed in between, it is better to cluster them into one level than to separate them.

Finally, there is the need for inter-level communication due to database concurrency control. Even though the current structured architecture enables a process executed within a particular level to access only data elements within that level, a transaction is nevertheless an entity that spans the boundary of processes. Control within a level that ensures mutual exclusion of processes within that level is not sufficient to guarantee transaction level serialiability. For example, if P11 and P12 are two processes executing at Levels 1 and 2 on behalf of transaction 1, and P21 and P22 are two processes executing at Levels 1 and 2 on behalf of transaction 2, and P11 is serialized to be before P21 at Level 1 while P12 is serialized to be after P22 at Level 2, then the overall sequence cannot produce an equivalent serialized execution of

these two transactions, and therefore the transaction level database consistency is violated.

To provide a formal treatment of this problem, we will first give a formal definition of control and data flow in the Functional Hierarchy and then proceed to relate the HDD approach to concurrency control to functional decomposition of Functional Hierarchy.

9.3.3.1 FORMAL DEFINITION OF CONTROL AND DATA FLOW IN INFOPLEX

OVERVIEW: We define a generalized multi-level database computer architecture as follows. A multi-level database computer is composed of n (n > 1) levels. The top level interfaces with facilities that communicate with the external world such as user terminals and telecommunication channels, and the bottom level implements a storage subsystem which communicates with all other levels in the database computer via a virtual storage interface.

Each level in the system has an Inter-level Request Queue (IRQ) from which the level obtains requests to be processed. Inter-level communication takes the form of inter-level requests (IR's); no shared memory is allowed. A level is physically implemented by a collection of processors and memory modules linearly linked together. The system is initialized with empty IRQ's. All processors at a level are 'idle'. Upon arrival of an IR in the IRQ of that level, an idle processor will

examine the nature of the IR and take some appropriate action. Specifically, if the IR is a *forward request*, a new process is created to process this IR, and the processor is dispatched to run the process. If the IR is a *return IR*, it is channeled to an existing process that this IR is addressed to. Therefore, a request may either be directed to an existing process at a level, or it may cause a new process to be generated. We assume that the arrival of a request is the only cause for a new process to be generated. Processing of a request at a level may cause new requests to be generated, which may then be deposited at the local IRQ or other IRQ's in the system. Processing of a request may also be suspended pending arrival of some other requests. Upon termination, a process destroys itself and releases the processor. Barring system failures, processing of a request will always terminate.

TRANSACTIONS AND REQUESTS: The system is driven by arrival of *user transactions*. Arrival of a transaction is signified by the deposit of a request from this transaction into the IRQ of the first level. In processing this request, new request may be generated and deposited at IRQ's. These new requests are, in general, labelled with the identity of the transaction, and are called request *on behalf of* that transaction. Therefore, all the requests processed in the system are identified with some transaction. There will be *no explicit inter-transaction communication* in the form of request passing.

OBJECTS: Each level i maintains a set of distinct objects that are to be shared by different transactions and requests. An object is a smallest stored unit of data at a level. It is through this set of objects that the effect of a request at a level can be communicated to another request (in addition to explicit request passing among those requests on behalf of the same transaction.) We shall denote the set of objects at the level i as O(i). Examples of object sets are virtual information definition table at the Virtual Information Level and security definition at the Security Level.

Object elements in O(i) can be examined, created, destroyed and modifed by processes at level i. (For the purpose of studying concurrency control, we will not include read-only objects at level i in the object set O(i).) Each object has some name(s) associated with it, and an operation will always name the object to be operated on. We will also denote the union of all object sets O(i) in the DBM to be O.

9.3.3.2 FORMAL CONDITIONS FOR CONSISTENCY

The object is the focal point of database concurrency control. The premise of the correctness of a concurrency control mechanism is that, if no concurrent processing of multiple transactions exist, that is, if a transaction is not started until the previous one is finished, then the database will be left in a consistent state at the end of processing of a transaction.

The notion of direct dependencies among transactions and that of a transaction dependency graph, developed in Chapter two, are directly applicable to each level of the multi-level DBM defined here. We rephrase these notions in our context as follows:

Definition. A ***transaction dependency graph*** of a schedule S at level i of a multi-level DBM is a digraph, denoted as $TG^{(i)}(S)$, where the nodes are the transactions in S and the arcs, representing ***direct dependencies*** between transactions, exist according to the following rules:

$t_2 \rightarrow t_1$ iff for some object element $d \in O(i)$,

(1) $w_1(d^j)$ and $r_2(d^j)$ are in S for some d^j, or

(2) $r_1(d^j)$ and $w_2(d^k)$ are in S for some d^j, d^k where d^j is the predecessor of d^k, or

(3) $w_1(d^j)$, $w_2(d^k)$ and $r_3(d^k)$ are in S and d^j is a predecessor of d^k.

Since by our definition of the multi-level DBM, the object sets O(i) and O(j), for levels i and j where $i \neq j$, do not intersect, therefore one can express the condition for overall serializability in terms of the acyclicity condition on the union of the transaction dependency graphs $TG^{(i)}(S)$, $i = 1, \ldots, n$. This is formally described in the following proposition:

Proposition 1

Consistency of a multi-level DBM is maintained if the global transaction dependency graph, defined as Union($TG^{(i)}(S)$) over all i = 1, ..., n, is acyclic.

To ensure the above condition, each level must first ensure that $TG^{(i)}(S)$ is acyclic, and a global mechanism must exist which ensures that if $t_1 \rightarrow t_2$ in $TG^{(i)}(S)$ for some i, then there exists no path $t_2 \rightarrow \ldots \rightarrow t_1$ in another $TG^{(j)}(S)$ for any $j \neq i$.

9.3.3.3 APPLYING THE HDD APPROACH TO CONCURRENCY CONTROL IN INFOPLEX

What has been stated is the condition for database transaction consistency to hold in a multi-level database computer as defined above. The basic notion is not very different from that of a non-duplicated distributed database.

It can be shown that either the two phase locking algorithm or the conventional timestamping algorithm, implemented at each level, will enforce global consistency. This follows directly from the fact that both algorithms impose a partial order on transaction dependencies and the multi-level DBM architecture does not alter this imposition. However, the multi-level multi-processor DBM environment does make certain aspects of the overhead involved in these conventional algorithms more costly. We will elaborate this point and show how the HDD approach of concurrency control can be exploited to improve this situation.

(1) Inter-level overhead: If the two-phase locking algorithm is used, the mechanisms used to prevent illegal interleaving are blocking and deadlocks. Blocking takes place when a request being processed at level i on behalf of transaction t_1 attempts to access an object in O(i) that is currently locked in an incompatible mode by another transaction t_2. (Note that the request on behalf of t_2 that locked this object may or may not be in existence any more at this point; however, t_2 must still be active somewhere at this point.) Therefore blocking is an activity performed strictly within a level and does not require inter-level communication. On the other hand, a deadlock may occur across levels. For example, a deadlock may occur if t_1 is blocked by t_2 at level i, t_2 blocked by t_3 at level j, and t_3 blocked by t_1 at level k. Distributed deadlocks must be detected and resolved using distributed deadlock detection algorithms (e.g., <Obermarck82>) which require a considerable amount of inter-level coordination.

The timestamping algoirthm does away with the distributed deadlock problem. However, if a transaction writes into the object sets of several levels, all these levels must be involved in performing atomic validation and commit of the transaction. This means that every object element that the transaction will write must be visited and, if the write request is validated, a temporary lock must held on the object until the transaction finishes writing and timestamping all elements and commits. If this operation involves more than one levels, it cannot be expected to be executed as fast as it could be if only one processor is

involved in performing it in a single critical section. Therefore the multi-level environment degrades the efficiency of the algorithm and much inter-level communication for the purpose of concurrency control may still be incurred.

To alleviate this problem, a guideline should be used that encourages the design of the Functional Hierarchy to cluster objects that are to be updated in a single transaction into object set(s) of one or only a few levels. In other words, this 'update cluster' property is one of the aspects that the functional decomposition methodology should achieve in order to reduce inter-level communication. Another motivation for accommodating the update cluster property in functional decomposition is that such decomposition can also benefit from the Hierarchical Timestamping Algorithm's capability of reducing intra-level inter-processor communication overhead. This latter point is elaborated in the following paragraphs.

(2) Intra-level overhead: As discussed in the beginning of this thesis, setting locks or leaving timestamps require the processor to first obtain the exclusive right to enter a critical section. This is required because lock tables or timestamp tables are themselves shared system resources. Alternatively, a processor can send a message to a dedicated process (or processor) which manages the tables, and wait for responses from the dedicated process. Either method requires inter-processor coordination within a level of the DBM to ensure that

the atomicity of the activity of setting locks or timestamping is enforced. This form of inter-processor data sharing is one of the factors that limit the effective number of processors a level of the DBM can have. (For a discussion of this concept of software lockout in a genreal multi-processor system, see <Madnick68>.)

Specifically, the timestamping algorithm involves the need to timestamp every data element accessed by a process executing on behalf of a particular transaction. A processor attempting to obtain the right to enter a critical section in order to read-timestamp a data element would be forced to wait if the critical section is at the same time being occupied by another processor attempting to perform timestamping. An algorithm that helps reducing this form of inter-processor interference would have the effect of increasing the effective level of multiprocessing within a level.

Therefore, a functional decomposition methodology and an accompanying transaction design that encourages the write sets of transactions in the system to be concentrated in one or a few levels would greatly reduce the inter-level synchronization overhead. In addition, if the object sets of all the levels in the system can be given a hierarchical order such that most of the transacitons in the system would read from object sets that are of higher hierarcical order than the object sets their write set is resided, then the Hierarchical Timestamping Algorithm can be profitably used to reduce the need to read timestamp data access

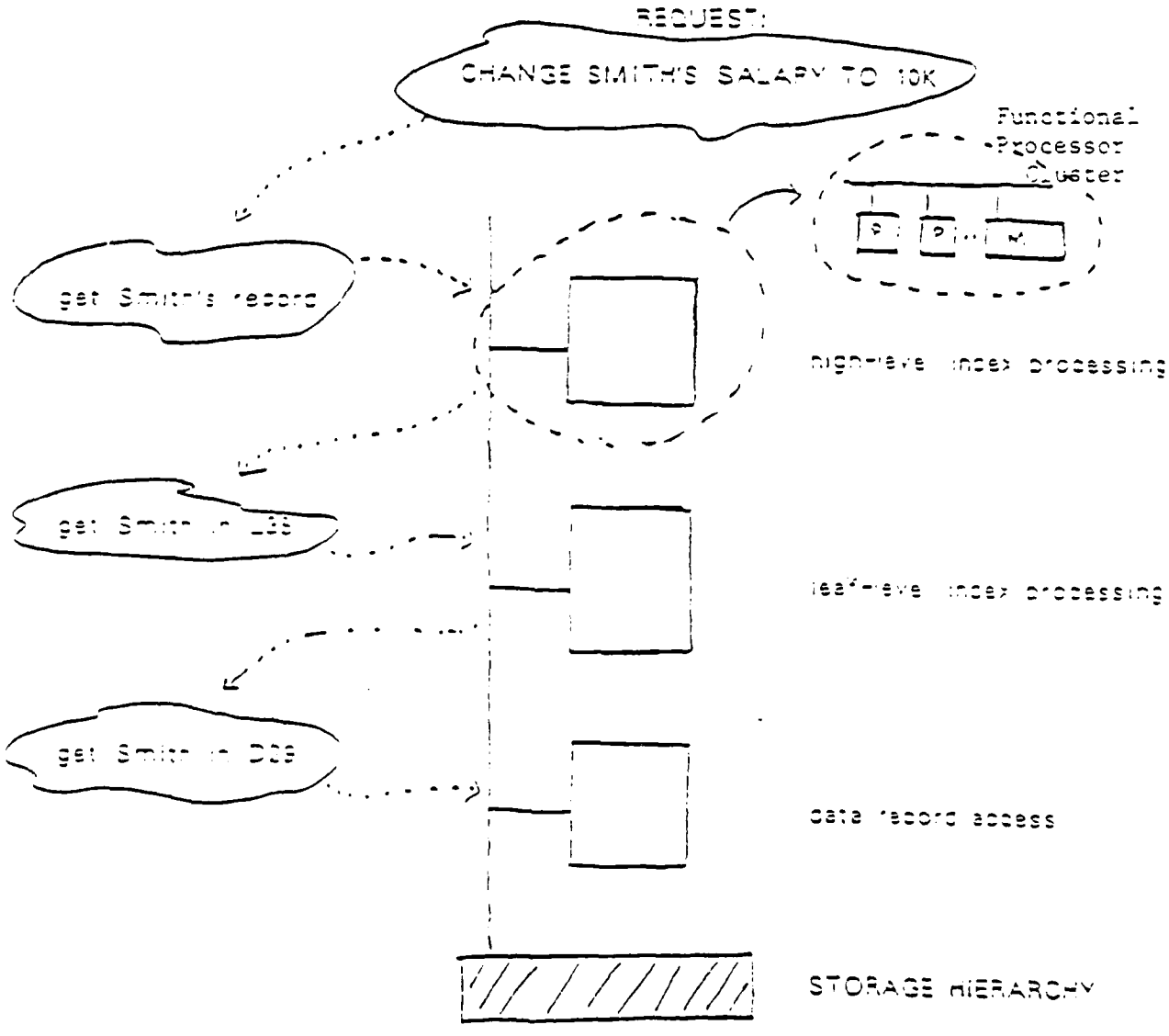


Figure 51. An example of INFOPLEX functional decomposition utilizing HDD.

requests for certain set of transactions, and therefore reduce the intra-level inter-processor interference.

An example illustrating the applicaiton of the HDD approach of concurrency control to the INFOPLEX database computer architecture is shown in Figure 51. In this example, the access method portion of the DBM is decomposed into three levels: the high-level index processing level (L1), the leaf-level index processing level (L2) and the data record access level (L3). Three classes of transactions are recognized by this portion of the system: key-updaters, non-key-updaters and index-balancing requets. L1 owns the object set of all high-level index pages (D_1), L2 the object set of all leaf-level index pages (D_2), and L3 the object set of all data records (D_3) A key-updater reads from D_1 and writes to D_2 and D_3. A non-key-updater reads from D_1 and D_2 and writes to D_3. Finally an index-balancing request would operate only in D_1. With this design, the key-updaters's read accesses to D_1 would not incur inter-processor overhead in L1. Similarly, the non-key-updaters' read accesses to D_1 and D_2 would not incur inter-processor overhead in L1 and L2.

In summary, this section has provided a formal treatment of the concurrency control problem in the Functional Hierarchy of the database computer INFOPLEX. It defines the formal conditions for database transaction level of consistency, and shows how this should be taken into consideration as one of the aspects that produce inter-level interfer-

ence. It then shows how the Hierarchical Timestamping Algorithm, by taking advantages of the need to structure the DBMS functionalities to reduce inter-level interference, can produce the effect of reducing intra-level inter-processor interferences. In general, the HDD perspective of concurrency control is expected to introduce the dimension of data decomposition into the formulation of a functional decomposition methodology.

10.0 SUMMARY AND FUTURE RESEARCH DIRECTIONS

10.1 SUMMARY

Database management systems are emerging as a crucial element of today's business operations. To fully exploit inherent parallelism in a computer system and to improve transaction response time, a DBMS must support multiple users at the same time, allowing multiple transactions to run in parallel. However, interleaving of the execution of multiple transactions may result in violation of database integrity. To prevent the latter from happening, a concurrency control facility must be included in a DBMS.

There is a growing concern over the efficiency of the concurrency control facility and its impact on the performance of a DBMS. Conventional approaches to concurrency control, taking serializability as the criterion of correctness, require that every read and write access request from a transaction be controlled by leaving a 'trace' (e.g., a lock or a timestamp) in the system. Setting locks or timestamping is an expensive operaton w ich not only incurs operatonal overhead but also produces some inter-transaction conflicts that could be otherwise avoided. The purpose of the current research is to seek ways to reduce

the overhead of synchronizing certain types of read accesses while achieving the goal of serializability.

To this end, a new approach of concurrency control for database management systems has been proposed. The technique makes use of a hierarchical database decomposition, a procedure which decomposes the entire database into a hierarchy of data segments based on the access pattern of the update transactions to be run in the system. A corresponding classification of the update transactions is derived where each transaction class is 'rooted' in one of the data segments. The heart of the new approach lies with differentiating the data accesses of an update transaction into three types: those accesses to the transaction's own root data segment, those to data segments higher than the transaction's own root data segment, and those to data segments lower than the root. The algorithm consists of three protocols: Protocol E for accessing data elements within the root segment; Protocol H for accessing higher data segments; and Protcol L for accessing lower data segments. When the data segment hierarchy consists of only one data segment, the hierarchical timestamp algorithm is reduced to the conventional multi-version timestamp algorithm.

The potential benefit of this algorithm stems from the usage of Protocol H (the 'cheap' protocol). When Protocol H is applicable, the algorithm enables read accesses to higher data segments to proceed without ever having to wait or to leave any trace of these accesses, thereby

reducing the overhead of concurrency control. A protocol (Protocol R) for handling ad-hoc read-only transactions in this environment is also devised, which does not require read-only transactions to wait or set any read timestamp. The proof of correctness of these algorithms in terms of their preservation of serializability is provided through a set of properties, lemmas and theorems which center around a new ordering concept called 'topologically follows.' These results are resported in Chapter three to Chapter six.

An implementation scheme for the proposed Hierarchical Timestamping Algorithm is described in Chapter seven. It addresses both the problems of multi-version database maintenance and timestamp management and attempts to achieve the goal of maximum parallelism within the concurrency control facility. In Chapter eight, the hierarchical database decomposition problem, i.e., the problem of how to find the optimal database partition and the data segment hierarchy that maximize the gain of using Protocol H, is formulated, and its complexity analyzed. The problem is shown to be NP-hard and a heuristic procedure is also proposed. Finally, in Chapter nine, the HDD approch is applied to three different application areas in which its effect on relieving database contention and on structuring databases and transactions so as to reduce concurrency control overhead is demonstrated.

10.2 FUTURE RESEARCH DIRECTIONS

The thrust of the research is more than proposing a new concurrency control algorithm. It demonstrates the potential benefit of exploiting the knowledge and the structure of the application systems to implement more efficient and more tailored concurrency control mechanism. Combining this realization and other aspects of this research, we propose the following future research directions.

(1) Additional studies for providing further evidences of the existence of hierarchical structural disciplines and applicability of the HDD approach: While Chapter nine has provided an exploratory study of the application of the HDD approach, further studies are needed in forms of case studies and performance modelling tools. In addition, specific database and transaction design guidelines that enable the application base to be broadened need to be formalized.

(2) Exploration of other algorithms that take advantages of the hierarchical structural discipline in a transaction processing system: The current algorithm is based on the idea that a concurrency control algorithm can assign an array of timestamps, rather than a single timestamp, to a transaction. This approach increases the flexibility of a timestamp-based algorithm. However, research into alternative timestamp-based algorithms which make use of a single timestamp and yet

still produce similar flexibility may prove to be a fruitful one. If this proves to be attainable, then different algorithms achieving the same goal of reducing synchronization overhead but of different performance features may be made available to system designers. Along the same vein, theoretical and practical investigations into whether similar flexibility can be built into lock-based algorithms will also be interesting undertakings.

(3) Exploration of other forms of structural disciplines that may render efficient concurrency control algorithms: Extending the original idea presented in the research of SDD-1, transaction analysis can continue to be used as a tool for discovering other forms of structural disciplines inherent in application studies. Moreover, current reasearch has but taken on the method of syntactic analysis of transactions to devise alternative algorithms. Semantic explorations, on the other hand, may yield even more powerful algorithms. This approach would be especially useful in systems where transaction types are very well formulated and understood, and large volume of transactions of a limited number of transaction types are repetitively executed.

(4) Multiprocessor database computer-specific studies: In order to fully utilize existing and emerging processor and memory technologies, the problem of how to intelligently organize components of a large-scale multiprocessor database computer so as to exploit the maximum level of concurrency expected in an application system presents a challenge all

by itself. The current research has only proposed one approach to dealing with the problem, focusing on the database concurrency control issue. The general philosophy advocated in this research is that there must be special structural disciplines, for reasons related or unrelated to increasing parallelism, that are to be incorporated into the multiprocessor database computer, and these disciplines present opportunities for one to design algorithms that are more intelligent than conventional ones. Of specific interests in this research direction are performance modeling of impact of concurrency control in a mutliprocessor system, and exploration of other methods that reduce inter-processor and inter-cluster communication.

BIBLIOGRAPHY

Astrahan76:

Astrahan, M.M. et al. System R: Relational approach to database management. ACM Trans. Database Syst. 1, 2 (June 1976) 25-35

Badal80:

Badal, D.Z. The analysis of the effects of concurrency control on distributed database system perforamcne. Proc. VLDB (1980)

Baer80:

Baer, J.L. et al. The two step commitment protocol: modelling, specification and proof methodology. Proc. 5th Intern. Conf. on Software Eng. (March 1980)

Bayer80:

Bayer, R., Heller, H., and Reiser, A. Parallelism and recovery in database systems. ACM Trans. Database Syst. 5, 2 (June 1980)

Bernstein79:

Bernstein, P.A., Shipman, D.W., and Wong, W.S. Formal aspects of serializability in database concurrency control. IEEE Trans. Software Eng. SE-5, 3 (May 1979)

Bernstein80:

Bernstein, P.A., Shipman, D.W., and Rothnie, J.B. Concurrency control in a System for Distributed Databases (SDD-1). ACM Trans. on Database Syst., 5, 1 (March 1980)

Bernstein81:

Bernstein, P.A., and Goodman, N. Concurrency control in distributed database systems. Computing Survey 13, 2 (June 1981).

Bernstein82:

Berstein, P.A., Goodman, N., and Hadzilacus Distributed database control and allocation. CCA Semi-annual technical report (July 1982)

Bernstein82b:

Bernstein, P.A. and Goodman, G.N., A sophisticate's introduction to database concurrency control. TR-19-82, Harvard University (September 1982)

BDM:

Private communication between William Frank and BDM technical staff. (May 1983)

BGS:

Private communication between the author and Mr. Allan Sarasohn, Manager of Technical Support, BGS Systems, August, 1983

Carey83:

Carey, M.J. Granularity hierarchies in concurrency control. Proceedings, ACM SIGACT-SIGMOD Symposium on Principles of Database Systems. (March 1983)

Ceri82:

Ceri, S. and Owicki, S. On the use of optimistic methods for concurrency control in distributed databases. 6th Berkeley Workshop on Distributed Database Management and Computer Networks. (Feb 1982)

Chamberlin73:

Chamberlin, D.D., Boyce, R.F., and Traiger, I.L. A deadlock-free scheme for resource locking in a data-base environment. IBM RJ 1329 (#20657) (Dec. 1973)

Chamberlin81:

Chamberlin, D.D. et al. A history and evaluation of system R. Comm. ACM 24, 10 (Oct. 1981)

Chan82:

Chan, A. et. al. The implementationof an integrated concurrency control and recovery scheme. Technical report CCA-82-01, Computer Corporation of America, Cambridge, Mass. (1982)

Chan82b:

Chan, A. and Gray, R. Implementing distributed read-only transactions. CCA Technical Report, 1982

Chesnais83:

Chesnais, A., Gelenbe, E. and Mitrani, I. On the modeling of parallel access to shared data. CACM, 26, 3 (March 1983)

Coffman71:

Coffman, Jr. E.G., Elphick, M.J., and Shoshani, A. System deadlocks. ACM Comput. Surv. 3, 2 (June 1971)

DeWitt79:

Dewitt, D.J. Direct - A multiprocessor organization for supporting relational database management systems. IEEE Trans. Computers C-28, 6 (June 1979)

Document-A:
IBISM Current Business Description, Citi Corp, New York (January 1983).

Document-C:
IBISM System Architecture Conceptual Model, Citi Corp, New York (Jan 1983)

Document-F:
IBISM Business Functions and Primitives, Citi Corp, New York (Jan 1983)

DuBourdieu82:
DuBourdieu, D.J. Implementation of distributed transactions. Proc., Berkeley Workshop on Distributed Database Management and Computer Networks. (1982)

Ellis77:
Ellis, C.S. Consistency and correctness of duplicate database systems. In Proc. 6th Symposium on Operating System Principles, West Lafayette, Ind. (1977) 67-84

Eswaran76:
Eswaran, K.P., Gray, J.N., Lorie, R.A., and Traiger, I.L. The notions of consistency and predicate locks in a database systems. Comm. ACM 19, 11 (Nov. 1976) , 624-634

Friedman80:
Friedman, L.J. IMS/VS BMP: uses and implications. Data Base/Data Communication Support Publication No. T1045.0-01A, Amdahl Corp. (July 1980)

Garcia-Molina82:
Garcia-Molina, H. and Wiederhold, G. Read-only transactions in a distributed database. ACM Trans. Database Syst. 7, 2 (June 1982)

Garey79:
Garey, M.R. and Johnson, D.S. Computers and Intractability: A Guide to the Theory of NP-Completeness. Freeman, San Francisco (1979)

Gardarin80:
Gardarin, G. An introduction to SABRE, a multiprocessor database computer. Publications Sabre (1980)

Goodman79:
Goodman, N., Bernstein, P.A., Wong, E., Christopher, R., and Rothnie, J.B. Query processing in SDD-1: a system for dis-

tributed databases. Computer Corporation of America, TR CCA-79-06, Cambridge, MA (Oct. 1979)

Goodman83:
Goodman, N., Suri, R. and Tay, Y.C. A simple analytical model for performance evaluation of exclusive locking in databases. Proc., Symposisum on Principles of Database Systems. (March 1983)

Gray75:
Gray, J.N., Lorie, R.A., and Putzolu, G.R. Granularity of locks in a shared data base. Proc. VLDB (1975)

Gray76:
Gray, J.N., Lorie, R.A., Putzolu, G.R., and Traiger, I.L. Granularity of locks and degrees of consistency in a shared data base. in Modelling in Data Base Management Systems, G.M. Nijssen. (ed.) North Holland Publishing Company (1976)

Gray78:
Gray, J.N. Notes on database operating systems. in O.S.: An Advanced Course. Vol 60, Lecture Notes in Computer Science, Spring-Verlag, N.Y. (1978)

Gray81:
Gray, J.N, Homan, P., Obermark, R. and Korth H. A strawman analysis of probability of waiting and deadlock. IBM Research Report, RJ 3066 (38112). (Feb 1981)

Hsiao77:
Hsiao, D.K., and Madnick, S.E. Database machine architecture in the context of information technology evolotion. Proc. VLDB (1977)

Hsiao79:
Hsiao, D.K., Ed. Collected readings on a database computer (DBC). Department of computer and Information Science, The Ohio State Univ., Columbus, Ohio (March 1979)

Hsiao80:
Hsiao, D.K. Database computers. In Advances in Computers, Academic Press, Vol. 19 (1980)

Hsu80:
Hsu M. A preliminary architectural design for the functional hierarchy of the INFOPLEX database computer. INFOPLEX Tech. Report No. 5, Sloan School of Management. (November 1980)

Hsu82:

Hsu, M. FSTV: The Software Test Vehicle for the Functional Hierarchy of the INFOPLEX database computer. INFOPLEX Tech. Report No. 9, Sloan School of Management. (December 1982)

Kedem80:

Kedem, Z and Silberchatz, A. Non-two phase locking protocols with shared and exclusive locks. Proc. VLDB. (1980)

King73:

King, P.F., and Collmeyer, A.J. Database sharing - An efficient mechanism for supporting concurrent processes. NCC (1973)

Korth82:

Korth, H. Deadlock freedom using edge locks. ACM Trans. on Database Systems, 7, 4. (Dec. 1982)

Kung79:

Kung, H.T. and Papadimitriou, C.H. An optimality theory of concurrency control for databases. ACM SIGMOD (1979)

Kung81:

Kung, H.T. and Robinson, J.T. Optimitic methods for concurrency control. ACM Trans. on Database Systems, 6, 2. (June 1981)

Lam79:

Lam, C.Y. and Madnick, S.E. Properties of Storage Hierarchy Systems with multiple page sizes and redundant data. ACM Trans. on Database Systems, 4, 3 (September 1979)

Lamport78:

Lamport, L. Time, clocks, and the ordering of events in a distributed system. Commun. ACM, 21, 7 (July 1978) 558-565

Lehman81:

Lehman, P.L. and Yao, S.B. Efficient locking for concurrent operations on B-trees. ACM Trans. on Database Systems, 6, 4. (Dec. 1981)

Lin82a:

Lin, W.K. and Nolte, J. Performance of two-phase locking. 6th berkeley workshop on Distributed Data Management and compter Networks. (Feb 1982)

Lin82b:

Lin, W.K. and Nolte, J. Communication delay and two-phase locking. Technical Report, Computer Corporation of America (1982)

Lin82c:
Lin, W.K. and Nolte, J. Performance of concurrency control algorithms using timestamps. Technical Report, Computer Corporation of America (1982)

Madnick68:
Madnick S.E. Multiprocessor software lockout. Proceedings of ACM National Conference. (1968)

Madnick79:
Madnick, S.E. The INFOPLEX database computer: concepts and directions. Proceedings IEEE Computer Conference. (Feb 1979)

Madnick83:
Madnick, S.E. and Hsu, M. Composite Information Systems: A strategy for system design. CISR Tech Rerpot, Sloan School of Management. (June 1983)

Mager80:
Mager, P.S., Goldber, R.P. A survey of some approaches to distributed data base and distributed file system architecture. BGS Systems, INC. Waltham, Mass. TR-80-001 (Jan 1980)

Maryanski80:
Maryanski, F.J. Backend database systems. ACM Comput. Surv. 12, 1 (March 1980)

Matteis79:
Matteis, R.J. The new back offic focuses on customer services. Harvard Business Review (March-April 1979)

Menasce80:
Menasce, D.A., Popek, G.J., and Muntz, R.R. A locking protocol for resource coordination in distributed databases. ACM Trans. Database Syst. 5, 2 (June 1980)

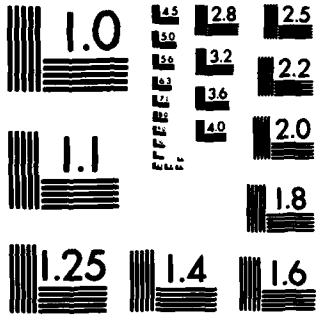
Menasce82:
Menasce, D.A. and Nakamishi, T. Performance Evaluation of two-phase commit protocol for DDBs. Proc., ACM Symp. on Princ. of Databases. (1982)

Mohan79:
Mohan, C. An analysis of the design of SDD-1: A system for distributed data bases. SDBEG-11, Univ. of Texas at Austin (April 1979)

Muntz77:
Muntz, R. and Krenz, G. Concurrency in database systems - a simulation study. Proc. ACM SIGMOD (Aug. 1977)

AD-A150 612 THE HIERARCHICAL DATABASE DECOMPOSITION APPROACH TO DATABASE CONCURRENCY..(U) ALFRED P SLOAN SCHOOL OF MANAGEMENT CAMBRIDGE MA CENTER FOR I.. M HSU DEC 84 4/4

UNCLASSIFIED CISR-TR-16 N00039-83-C-0463 F/G 9/2 NL



MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

Obermarck82:
Obermarck, R. Distributed deadlock detection algorithm. ACM Trans. on Database Systems, 7, 2. (June 1982)

Papadimitriou79:
Papadimitriou, C.H. The serializability of concurrent database updates. Journal of ACM 26, 4 (Oct 1979)

Papadimitriou82:
Papadimitriou, C.H. and Kanellakis, P.C. On concurrency control by multiple versions. Proc. 1982 ACM SIGACT-SIGMOD Symp. on Principles of Database Syst. (March 1982)

Reed78:
Reed,D.P. Naming and synchronization in a decentralized computer system. Ph.D. dissertation, Dept. of Electrical Engineering and Computer Science, M.I.T., Cambridge, Mass. (September 1978)

Reef79:
Reef, R.K. Citibank NA and distributed data processing. In Managing the Distribution of Data Processing, Infotech International, Marlow, England (1979)

Ries77:
Ries, D. and Stonebraker M. Effects of locking granularity in a DBMS. ACM Trans. on Database Systems, 2, 3 (Sept 77).

Ries79:
Ries, D. and Stonebraker, M. Locking granlarity revisited. ACM Trans. on Database Systems, 4, 2 (June 79)

Rosenkrantz78:
Rosenkrantz, D.J., Stearns, R.E., and Lewis II, P.M. System level concurrency control for distributed database systems. ACM Trans. Database Syst. 3, 2 (June 1978)

Rosenkrantz82:
Rosenkrantz, D.J., Stearns, R.E. and Lewis II, P.M. consistency and Serializability in concurrent database systems. To appear in SIAM Journal of Computing (1982)

Rypka79:
Rypka, D.J., and Lucido, A.P. Deadlock detection and avoidance for shared logical resources. IEEE Trans. Software Eng. SE-5, 5 (Sept. 1979)

Silberschatz80:
Silberschatz, A., and Kedem, z. Consistency in hierarchical database systems. Journal of ACM 27, 1 (Jan 1980) , 72-80

Spitzer76:
Spitzer, J.F. Performance prototying of data managmenet applications. Proc. ACM Annual Conf., Houston, Tex. (Oct, 1976)

Stearns76:
Stearns, R.E., Lewis II. P.M. andRosenkrantz, D.J. Concurrency Control for database systems. Proc. 17th Annual Symp. on Foundations of Computer Science, Houston, Tx. (Oct, 1976)

Stearns81:
Stearns, R. and Rosenkrantz, D. Distributed database concurrency control using before-values. ACM SIGMOD Conference Proceeding (1981)

Stonebraker76:
Stonebraker, M. Wong, E., Kreps, P. and Held, G. The design and implementation of INGRES. ACM Trans. on Database Systems, 1,3 (1976)

Stonebraker80:
Stonebraker, M. Retrospection on a database system. ACM Trans. Database Syst. 5, 2 (June 1980) , 225-240

Thomas79:
Thomas, R.H. A majority consensus approach to concurrency control for multiple copy databases. ACM Trans. Database Syst. 4, 2 (June 1979) ,180-209

To82:
To, T. SHELL: a simulator for the software test vehicle of the INFOPLEX database computer. INFOPLEX Tech. Report No. 8, Sloan School of Management, M.I.T (Jan 1982)

Ullman82:
Ullman, J.D. Principles of Database Systems. Computer Science Press, Rockville, Maryland (1982)

Viemont82:
Viemont, Y.H. and Gardarin, G.J. A distributed concurrency control algorithm based on transaction commit ordering. Procedding of Fault Tolerance Computer Systems, Santa Monica, Cal. (June 1982)

Verhofstad78:
Verhofstad. Recovery techniques for database systems. ACM Comput. Surv. (June 1978)

END

FILMED

3-85

DTIC